С. Дасгупта, Х. Пападимитриу, У. Вазирани

Алгоритмы

Перевод с английского А. С. Куликова под редакцией А. Шеня

Москва
Издательство МЦНМО
2014

УДК 510.5
ББК 22.18я73
Д20

Дасгупта С. и др.
Д20 Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани; Пер. с англ. под ред. А. Шеня. — М.: МЦНМО, 2014. — 320 с.

ISBN 978-5-4439-0236-4

В этой книге, предназначенной для студентов математических и программистских специальностей (начиная с младших курсов), подробно разбираются основные методы построения и анализа эффективных алгоритмов. Она основана на лекциях авторов в университетах Сан-Диего и Беркли. Выбор материала не вполне стандартный (скажем, о сортировке и структурах данных, связанных с хранением упорядоченных множеств в сбалансированных деревьях, не говорится, зато обсуждаются линейное программирование и даже квантовые вычисления). Авторы старались выделить основные идеи и излагать доказательства наглядно, не злоупотребляя формализмом, но и не жертвуя математической строгостью; оригинальный подход авторов делает книгу интересной не только студентам, но и опытным преподавателям. Каждый раздел снабжён упражнениями.

ББК 22.18я73

ISBN 978-0073523408 (англ.)
ISBN 978-5-4439-0236-4

Оглавление

Предисловие

Эта книга возникла из записок лекций, которые авторы читали в последние десять лет [английское издание вышло в 2008 году] в университетах Беркли и Сан-Диего для студентов младших курсов. За это время курс сильно изменился. Отчасти это было связано с тем, что студенты редко имеют опыт строгих рассуждений и нам пришлось с этим считаться, отчасти — с эволюцией самого предмета лекций. Выбор материала определялся логикой рассказа, и мы не претендуем на энциклопедичность и отступили от традиций, включив некоторые темы. Мы старались выделять идеи, лежащие в основе доказательств, и не злоупотреблять формализмом — в надежде, что это сделает математическую строгость более привлекательной.

Следуя истории, мы решили начать курс (часть I) с алгоритмов, работающих с числами. Начав со знакомых понятий (простота, разложение на множители), мы затем рассматриваем алгоритмы умножения, сортировки и поиска медианы, основанные на приёме «разделяй и властвуй». Также мы рассматриваем алгоритм быстрого преобразования Фурье. Далее (часть II) мы переходим к традиционным темам вводного курса: структурам данных и графам. Здесь мы стремимся показать, как совсем короткие алгоритмы (несколько строк кода) позволяют решать довольно сложные задачи. При желании следовать традиционной схеме курса можно начать с части II (которая не зависит от предыдущего материала, не считая «Пролога») и отложить материал части I «на потом» (если останется время).

В частях I и II мы излагаем некоторые приёмы, полезные для задач определённых классов («разделяй и властвуй», жадные алгоритмы). В части III мы переходим к более общим методам и разбираем динамическое программирование (постаравшись изложить его более понятно, чем это часто делается) и линейное программирование (симплекс-метод, двойственность, представление комбинаторных задач в виде задач линейного программирования). Наконец, в части IV мы обсуждаем алгоритмически трудные задачи (NP-полнота, эвристические алгоритмы, квантовые вычисления). В итоге мы замыкаем круг, вернувшись к задаче разложения на множители и разобрав квантовый алгоритм Шора.

Параллельные основному тексту врезки посвящены истории развития области, практическим приложениям (особенно связанным с интернетом) и математическим отступлениям.

Глава 0
Пролог

Куда ни посмотри — везде компьютеры. Торговля, развлечения, производство, медицина — всё это выглядело бы совсем иначе без компьютеров. Даже генералы и президенты понемногу учатся ими пользоваться. Из двух технологических прорывов, сделавших все эти чудеса возможными, один очевиден — это колоссальный прогресс в области технологии разработки и производства микросхем, которые становятся меньше, быстрее и экономичнее с каждым годом.

Но наша книга посвящена другому прорыву — изобретению эффективных алгоритмов. Он не столь заметен, но не менее важен и не менее увлекателен, так что бросайте свои дела — и вперёд, за нами!

0.1. Книги и алгоритмы

В 1448 году в немецком городе Майнце ювелир по имени Иоганн Гутенберг придумал, что можно печатать книги, набирая текст из отдельных литер. Мрачному средневековью пришёл конец, науки и искусства расцвели, затем началась индустриальная революция и всё такое. Многие историки видят в этом заслугу Гутенберга — и действительно, мир был бы совсем другим, если книги нужно было бы переписывать вручную. Но другие отводят главную роль менее заметному изобретению — возникновению *алгоритмов*.

Сейчас все с младых ногтей приучены к десятичной системе. А между тем Гутенберг записал бы 1448 год как MCDXLVIII. Сможете ли вы вычислить сумму MCDXLVIII + DCCCXII? а произведение этих чисел? Скорее всего Гутенберг, как человек умный и образованный, мог складывать и вычитать небольшие числа на пальцах; для более серьёзных вычислений он должен был бы обратиться к специалистам, умеющим считать на абаке.

Десятичная система, изобретённая в Индии около 600 года н. э., произвела революцию в технологии вычислений: она позволила с помощью всего лишь десяти символов компактно записывать большие числа и выполнять арифметические операции как последовательность простых шагов. Несмотря на все эти преимущества, десятичная система распространилась по миру не сразу — как всегда, сказались расстояния, языковые и культурные барьеры. Большую роль в её распространении сыграла книга, написанная по-арабски в IX веке. Её автор жил в Багдаде, звали его аль-Хорезми, и в ней он описал методы сложения, умножения и деления в десятичной системе (и даже вычисления квадратных корней и цифр числа π). Методы эти требовали аккуратного («механического») выполнения чётко описанных действий и гарантировали

получение правильного результата — короче говоря, они были *алгоритмами*, как мы сказали бы сейчас. Но само слово *алгоритм* появилось много лет спустя и произошло как раз от имени аль-Хорезми.

С этого времени десятичная система и основанные на ней численные алгоритмы играют центральную роль. Без них не было бы ни науки, ни техники, ни быстрого развития промышленности и торговли. А когда появились компьютеры, позиционная система (правда, двоичная, а не десятичная) легла в основу представления чисел в их памяти. Новые алгоритмы самого разного рода появляются каждый год, всё больше расширяя возможности компьютеров — и меняя наш мир до неузнаваемости.

0.2. Вычисление чисел Фибоначчи

В последовательности Фибоначчи

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

каждый член равен сумме двух предыдущих:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & n > 1, \\ 1, & n = 1, \\ 0, & n = 0. \end{cases}$$

Это одна из самых знаменитых последовательностей — почти как последовательность степеней двойки. Числа Фибоначчи растут *почти* так же быстро, как степени двойки: например, F_{30} больше миллиона, а в десятичной записи F_{100} уже 21 цифра. Число F_n равно примерно $2^{0{,}694n}$ (см. упражнение 0.3).

Как вычислять числа Фибоначчи?

Экспоненциальный алгоритм

Буквально следуя рекурсивному определению, получаем следующий алгоритм.

```
функция Fib1(n)
если n = 0:   вернуть 0
если n = 1:   вернуть 1
вернуть Fib1(n − 1) + Fib1(n − 2)
```

Написав алгоритм, мы должны спросить себя:

1. Корректен ли он?

2. Каково время его работы (в зависимости от n)?

3. Существует ли более быстрый алгоритм?

В нашем случае первый вопрос не имеет особого смысла, поскольку алгоритм повторяет определение. Для ответа на второй вопрос обозначим через $T(n)$ время работы (число операций) алгоритма Fib1 на входе n. Ясно, что для $n \leqslant 1$ значение $T(n)$ не больше двух, а далее $T(n) = T(n-1) + T(n-2) + 3$

(два рекурсивных вызова, две проверки значения n и сложение). Сравнив это неравенство с определением F_n, мы видим, что $T(n) \geqslant F_n$. Таким образом, время работы алгоритма растёт *экспоненциально* с ростом n, что означает, что он практически бесполезен на практике. Например, для вычисления F_{200} понадобится около 2^{138} операций. На это даже суперкомпьютеру, выполняющему 40 триллионов операций в секунду, понадобится 2^{92} секунд.

Есть такое наблюдение, называемое *законом Мура*: каждый год компьютеры становятся быстрее примерно в 1,6 раз. Представим себе, что такое продлится ещё некоторое время и посмотрим, много ли чисел Фибоначчи нам удастся вычислить. Время работы алгоритма Fib1 растёт примерно как $2^{0{,}694n} \approx 1{,}6^n$. Если сегодня компьютер позволяет вычислить F_{100}, то через год мы вычислим F_{101}, через два года — F_{102}. И так далее, по одному числу в год.

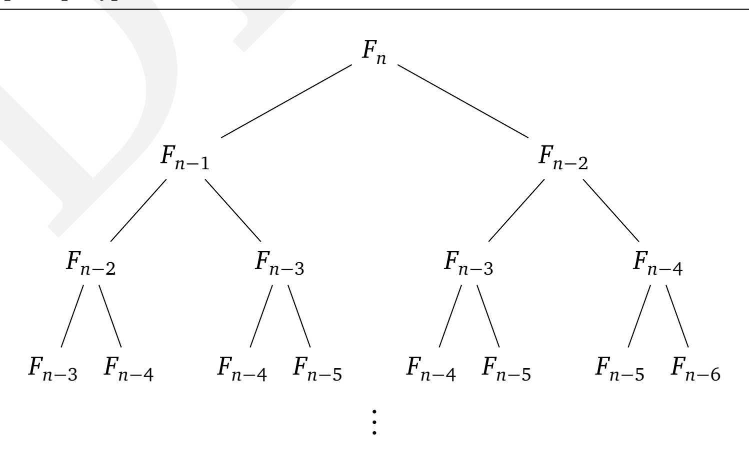
Итак, наш алгоритм корректен, но безнадёжно неэффективен. Существует ли более быстрый алгоритм?

Полиномиальный алгоритм

Попытаемся понять, почему же алгоритм Fib1 такой медленный. На рис. 1 показано дерево рекурсивных вызовов. Видно, что алгоритм много раз вычисляет одно и то же. Разумнее было бы сохранять промежуточные результаты (значения $F_0, F_1, \ldots, F_{n-1}$):

```
функция Fib2(n)
если n = 0:   вернуть 0
создать массив f[0...n]
f[0] ← 0, f[1] ← 1
для i от 2 до n:
   f[i] ← f[i − 1] + f[i − 2]
вернуть f[n]
```

Рис. 1. Дерево рекурсивных вызовов



Как и в случае предыдущего алгоритма, корректность здесь очевидна. Каково же время работы? Цикл повторяется $n-1$ раз, и время работы алгоритма Fib2 *линейно по* n. Переход от экспоненциального к полиномиальному (в данном случае линейному) времени работы радикально меняет дело. Теперь F_{200} и даже F_{200000} можно вычислить без труда[1]: секрет успеха — в правильном алгоритме.

Более детальный анализ

До сих пор мы оценивали число операций в алгоритме и неявно предполагали, что все операции занимают одно и то же время. Это удобно, хотя на деле каждая операция разбивается процессором на ещё более мелкие шаги и количество и длительность этих шагов могут быть разными.

Есть в нашем анализе и более существенное упрощение: мы считали сложение за один шаг, не учитывая размеры слагаемых. Это разумно, если числа помещаются в один регистр (32 или 64 бита). Однако в двоичной записи n-го числа Фибоначчи около $0{,}694n$ битов, и они очень скоро перестают помещаться. Арифметические операции на числах произвольной длины не могут быть выполнены за один шаг. Поэтому нужно более детально оценить время работы алгоритма.

В главе 1 мы увидим, что сложение двух n-битовых чисел требует времени, пропорционального n (сложение в столбик). Значит, алгоритм Fib1, выполняющий около F_n сложений, требует около nF_n *элементарных операций*. Для алгоритма Fib2 время пропорционально n^2, что по-прежнему не так много.

Может быть, есть ещё более быстрый алгоритм? Оказывается, что да: см. упражнение 0.1.

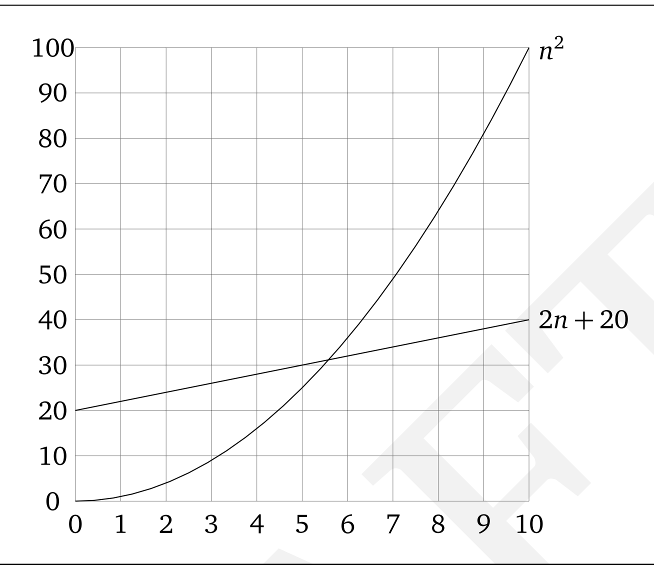
0.3. *O*-символика

Оценивая примерное время работы алгоритма, важно понимать, чем мы пренебрегаем, а за чем следим (чтобы не пропустить главное и не копаться в несущественных деталях). Распространённый подход — оценивать время работы алгоритма (как функцию от размера входа) с точностью до ограниченных множителей. При этом ошибка в два раза или даже в тысячу раз считается допустимой, в то время как разница между n и n^2 считается существенной.

При таком подходе нет надобности разбираться в длительности элементарных операций, хотя в реальных процессорах разные команды разлагаются в последовательность микрокоманд разной длины и занимают разное время. Важно только, что время любой операции ограничено некоторой константой. Другое следствие того же подхода: если, скажем, алгоритм выполняет $5n^3+4n+3$ элементарных операций на входе размера n, мы можем отбросить слагаемые $4n$ и 3 (которые при бóльших n малы по сравнению с $5n^3$). Более того, мы можем отбросить и множитель 5 в старшем слагаемом (через несколько лет компьютеры станут в пять раз быстрее) и сказать, что время

[1] О разнице между полиномиальным и экспоненциальным ростом см. историю в главе 8.

Рис. 2. Сравнение квадратичной и линейной функций



работы алгоритма есть $O(n^3)$ (произносится как «о большое от эн в кубе»). Сейчас мы объясним смысл этих O-обозначений.

> Пусть даны две функции $f(n)$ и $g(n)$ натурального аргумента n, значениями которых являются положительные действительные числа. Говорят, что $f = O(g)$ («f растёт не быстрее g»), если *существует такая константа* $c > 0$, *что* $f(n) \leqslant c \cdot g(n)$ *для всех* $n \in \mathbb{N}$.

Другими словами, $f = O(g)$ означает, что отношение $f(n)/g(n)$ ограничено сверху некоторой константой.

Запись $f = O(g)$ можно читать как «$f \leqslant g$ с точностью до константы». При этом, скажем, $10n = O(n)$. Несмотря на такую погрешность при малых n, мы можем сравнивать поведение функций при больши́х n. Например, пусть один алгоритм требует $f_1(n) = n^2$ шагов, а второй — $f_2(n) = 2n + 20$ (см. рис. 2). Какой из них лучше? Это, конечно, зависит от значения n: первый лучше второго при $n < 5$, но сильно хуже при большом n. В наших обозначениях можно сказать, что $f_2 = O(f_1)$, но $f_1 \neq O(f_2)$. В самом деле, отношение

$$\frac{2n+20}{n^2} = \frac{2}{n} + \frac{20}{n^2} \leqslant 22$$

ограничено, а

$$\frac{n^2}{2n+20} \geqslant \frac{n^2}{22n} = \frac{n}{22}$$

— нет.

Пусть есть и третий алгоритм, время работы которого равно $f_3(n) = n + 1$. Лучше ли он второго? Да, конечно, но всего лишь в несколько раз, и $f_2 = O(f_3)$

и наоборот. В самом деле,

$$\frac{f_2(n)}{f_3(n)} = \frac{2n+20}{n+1} \leqslant 20 \quad \text{и} \quad \frac{f_3(n)}{f_2(n)} \leqslant 1.$$

Обозначение $O(\cdot)$ можно считать аналогом $\leqslant$. Аналоги для $\geqslant$ и $=$ такие:

$f = \Omega(g)$ (f растёт не медленнее g, с точностью до константы) означает $g = O(f)$;

$f = \Theta(g)$ (f и g имеют одинаковый порядок роста) означает, что $f = O(g)$ и $g = O(f)$.

Для рассматриваемых нами функций $f_2 = \Theta(f_3)$ и $f_1 = \Omega(f_3)$.

Благодаря O-символике мы можем заменить $3n^2 + 4n + 5$ на $\Theta(n^2)$, пренебрегая остальными слагаемыми. Вот несколько общих правил такого рода замен:

1. Постоянные множители можно опускать. Например, $14n^2$ можно заменить на n^2.

2. n^a растёт быстрее n^b для $a > b$. Например, в присутствии слагаемого n^2 можно пренебречь слагаемым n.

3. Любая экспонента растёт быстрее любого многочлена (полинома). Например, 3^n растёт быстрее n^5.

4. Любой полином растёт быстрее любого логарифма. Например, n (и даже $\sqrt{n}$) растёт быстрее $(\log n)^3$; мы не указываем основания логарифма, поскольку замена основания приводит лишь к умножению логарифма на константу. По тем же причинам n^2 растёт быстрее $n \log n$.

Не подумайте, что на практике никого не интересуют постоянные множители: ускорение алгоритма вдвое может быть весьма трудоёмким, но необычайно выгодным. Но для начала разумно оценить ситуацию в первом приближении, и O-символика тут весьма полезна.

Упражнения

0.1. Выясните, верно ли, что $f = O(g)$ и $f = \Omega(g)$ (возможно, верно и то, и другое, тогда $f = \Theta(g)$) для следующих функций:

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$n^{1/2}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)^2$
(d)	$n \log n$	$10n \log(10n)$
(e)	$\log(2n)$	$\log(3n)$
(f)	$10 \log n$	$\log(n^2)$
(g)	$n^{1.01}$	$n \log^2 n$
(h)	$n^2 / \log n$	$n(\log n)^2$

(i)	$n^{0.1}$	$(\log n)^{10}$
(j)	$(\log n)^{\log n}$	$n/\log n$
(k)	$\sqrt{n}$	$(\log n)^3$
(l)	$n^{1/2}$	$5^{\log_2 n}$
(m)	$n2^n$	3^n
(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^{n} i^k$	n^{k+1}

Мы не указываем основание логарифма в тех случаях, когда оно не играет роли.

0.2. Покажите, что для произвольной константы $c > 0$ функция $g(n) = 1 + c + c^2 + \ldots + c^n$ есть

(a) $\Theta(1)$, если $c < 1$;

(b) $\Theta(n)$, если $c = 1$;

(c) $\Theta(c^n)$, если $c > 1$.

Другими словами, в сумме убывающей геометрической прогрессии можно оставить лишь первый член; возрастающей — последний, а постоянной — количество членов.

0.3. Напомним, что в последовательности Фибоначчи $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$. Убедитесь, что у данной последовательности экспоненциальный порядок роста, получив такие оценки:

(a) Докажите по индукции, что $F_n \geqslant 2^{0,5n}$ при $n \geqslant 6$.

(b) Найдите константу $c < 1$, для которой $F_n \leqslant 2^{cn}$ при всех $n \geqslant 0$.

(c) Для какого максимального c вы можете доказать, что $F_n = \Omega(2^{cn})$?

0.4. Можно ли ускорить алгоритм Fib2 (с. 9)? Оказывается, что да, если использовать матрицы. Запишем равенства $F_1 = F_1$ и $F_2 = F_0 + F_1$ с помощью матриц:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Аналогичным образом,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix},$$

и вообще

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Значит, для вычисления F_n достаточно возвести матрицу $X = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ в степень n.

(a) Покажите, что для перемножения двух матриц размера 2×2 достаточно четырёх сложений и восьми умножений (чисел).

Но сколько умножений матриц потребуется для вычисления X^n?

(b) Покажите, что $O(\log n)$ матричных умножений будет достаточно. (Подсказка: подумайте, как вычислить X^8.)

В итоге количество арифметических операций для нового алгоритма (назовём его Fib3) есть всего $O(\log n)$, что гораздо меньше, чем $O(n)$ для Fib2.

Оптимист мог бы обрадоваться и сказать, что мы снова экспоненциально ускорили алгоритм: теперь вместо $O(n)$ операций у нас всего $O(\log n)$ операций. Но радость эта преждевременна: подвох в том, что новый алгоритм не только складывает, но и перемножает числа, а умножение больших чисел — более медленная операция, чем сложение. Как мы уже видели, с учётом сложности арифметических операций время работы алгоритма Fib2 есть $O(n^2)$.

(c) Покажите, что все числа, встречающиеся в процессе работы алгоритма Fib3, имеют длину $O(n)$.

(d) Пусть $M(n)$ — время умножения двух n-битовых чисел. Ясно, что $M(n) = O(n^2)$ (школьный метод умножения «в столбик», рассматриваемый в главе 1, имеет как раз такое время работы). Покажите, что время работы алгоритма Fib3 есть $O(M(n) \log n)$.

(e) Докажите более сильную оценку $O(M(n))$ на время работы алгоритма Fib3. (Подсказка: длины умножаемых чисел удваиваются с каждым возведением в квадрат.)

Будет ли алгоритм Fib3 быстрее, чем Fib2? Это зависит от того, сможем ли мы умножать n-битовые числа быстрее, чем за $O(n^2)$ (ответ мы узнаем в главе 2).

В заключение приведём формулу для чисел Фибоначчи:

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

Эта формула позволяет свести вычисление F_n к возведению двух чисел в степень n. Однако числа эти иррациональны и перемножать их с достаточной точностью — непростая задача.

Знающие линейную алгебру могут заметить, что на самом деле алгоритм Fib3 как раз и возводит эти два числа в степень n. (Подсказка: подумайте, каковы собственные числа матрицы X.)

Глава 1

Числовые алгоритмы

В данной главе мы увидим, как сильно различаются две (очень похожие на первый взгляд) задачи:

- Разложение на множители: представить данное число N как произведение простых.
- Проверка на простоту: выяснить, является ли данное число N простым.

Разложение на множители является очень сложной задачей. Несмотря на усилия учёных всего мира, самые быстрые алгоритмы разложения числа N на простые множители по-прежнему требуют экспоненциального времени. (Показатель экспоненты — количество битов в записи числа, точнее, некоторый корень из этого количества.) В то же время, как мы скоро увидим, проверить простоту числа N можно довольно быстро. И, что самое интересное, на этом разрыве между двумя родственными задачами основаны современные технологии безопасного обмена информацией.

Попутно мы разберём алгоритмы для различных операций с числами. Мы начнём с арифметических операций (что даже исторически оправдано: *алгоритмами* изначально назывались методы решения как раз таких задач).

1.1. Элементарная арифметика

1.1.1. Сложение

Когда школьника учат складывать два числа столбиком, ему не объясняют, почему этот способ годится (а если и пытаются, то вряд ли это доходит). Посмотрим на него немного подробнее.

Ключевую роль играет такое наблюдение: *сумма любых трёх цифр будет однозначным или двузначным числом.* (Действительно, такая сумма не больше $9 + 9 + 9 = 27$.) Это же верно и для любого основания $b \geqslant 2$ (упражнение 1.1). Скажем, в двоичной системе счисления сумма любых трёх цифр не превосходит 3, то есть 11 в двоичной записи.

Почему это важно? Когда мы складываем две цифры в разряде единиц, может появиться цифра в разряде десятков («один пишем, один в уме»). Тогда в разряде десятков мы должны складывать уже три цифры, но всё равно получится двузначное число и перенос потребуется только в следующем разряде (сотен). Складывая три цифры в разряде сотен, мы получаем разряд сотен суммы и цифру переноса, и так далее.

Основания и логарифмы

Популярность десятичной системы, видимо, связана с тем, что у нас 10 пальцев. Древние майя использовали основание 20 (тоже естественно, если ходить босиком). А в компьютерах числа представляются в двоичной системе счисления.

Сколько нужно цифр, чтобы записать целое число $N > 0$ в системе счисления с основанием b? При помощи k цифр (от 0 до $b-1$ каждая) можно представить числа от 0 до $b^k - 1$. Например, с помощью трёх десятичных цифр можно записать числа до $999 = 10^3 - 1$. Отсюда легко получить ответ: для записи числа N в b-ичной системе счисления нужно $\lceil \log_b(N+1) \rceil$ разрядов (примерно $\log_b N$, если допустить ошибку плюс-минус один разряд).

Как меняется число разрядов при изменении основания системы счисления? Вспомним, что $\log_b N = (\log_a N)/(\log_a b)$. Значит, при переходе от основания b к основанию a размер записи (примерно) умножается на $\log_a b$. Поэтому для записи числа N нужно $O(\log N)$ цифр, какова бы ни была система счисления. По этой причине внутри $O(\cdot)$ основание логарифма обычно не указывается. Когда мы используем логарифм без основания вне $O(\cdot)$, мы имеем в виду двоичный логарифм.

Функция $\log N$ встретится нам не раз. Её можно описать так:

1. По определению $\log N$ — это степень, в которую нужно возвести 2, чтобы получить N.

2. Число N нужно уполовинить $\log N$ раз, чтобы получить 1 или меньше (точнее говоря, $\lceil \log N \rceil$ раз). (Это наблюдение позволяет оценить число повторений цикла, в котором некоторая величина убывает по крайней мере вдвое.)

3. $\log N$ — количество битов в двоичной записи числа N (точнее говоря, в ней $\lceil \log(N+1) \rceil$ битов).

4. $\log N$ — глубина полностью заполненного двоичного дерева с N вершинами (точнее говоря, она равна $\lfloor \log N \rfloor$).

5. $\log N = \Theta\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}\right)$ (упражнение 1.5).

Для примера сложим числа 53 и 35 в двоичной системе счисления (где они записываются как 110101 и 100011):

перенос:	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

Мы не будем приводить псевдокод сложения столбиком, а сразу перейдём к анализу его эффективности. *Сколько времени потребуется, чтобы сложить два числа x и y*? Точнее говоря, как число операций зависит от размера входных данных (количества битов в записи x и y)?

Подобные вопросы мы будем задавать себе по поводу всех рассматриваемых алгоритмов.

Допустим, что x и y — числа из n битов (или меньше). Тогда в двоичной записи $x + y$ будет не более $n + 1$ битов. На вычисление каждого бита суммы уходит ограниченное константой время. Общее время *линейно* ($c_0 + c_1 n$ с какими-то константами c_0, c_1). Эти константы нас не интересуют, и время работы мы записываем просто как $O(n)$.

Оценив время работы имеющегося алгоритма, мы должны спросить себя: существует ли более быстрый алгоритм? В данном случае ответ прост: существенно более быстрого алгоритма существовать не может, поскольку уже для чтения входных битов и записи ответа требуется $O(n)$ операций. Значит, наш алгоритм сложения оптимален с точностью до мультипликативной константы.

Могут спросить — к чему все эти разговоры? Разве современные компьютеры не складывают два числа за один шаг? Есть два ответа. Во-первых, действительно, за один шаг можно сложить два числа, которые помещаются в машинное слово (обычно 32 или 64 бита). Однако, как мы увидим, часто необходимо работать с гораздо более длинными числами, и их волей-неволей приходится разбивать на части. Во-вторых, хотя сложение машинных слов производится одной командой, внутри процессора эта команда состоит из многих операций с отдельными битам. И число таких *битовых операций* существенно, поскольку от него зависит количество функциональных элементов (транзисторов, проводов и т. д.), необходимых для реализации алгоритма «в железе».

1.1.2. Умножение и деление

Перейдём к умножению. Школьный метод умножения двух чисел x и y «в столбик» заключается в том, чтобы умножить x на каждую цифру числа y и сложить результаты с соответствующим сдвигами. Для примера перемножим в двоичной системе числа 13 и 11 (соответственно 1101 и 1011):

				1	1	0	1	
			×	1	0	1	1	
				1	1	0	1	(1101 умножить на 1)
+			1	1	0	1		(1101 умножить на 1, сдвинутое на 1)
		0	0	0	0			(1101 умножить на 0, сдвинутое на 2)
	1	1	0	1				(1101 умножить на 1, сдвинутое на 3)
1	0	0	0	1	1	1	1	(143 в двоичной системе счисления)

Дело упрощается тем, что в двоичной записи каждая цифра в x есть либо нуль (тогда в произведении нуль), либо 1 (и тогда достаточно сдвинуть y), так что никакой таблицы умножения помнить не надо. Корректность этого алгоритма мы просим доказать в упражнении 1.6.

Каково же время его работы? Если в двоичной записи чисел x и y по n битов, то будет n промежуточных результатов длины не более $2n$ (учитывая нули на конце после сдвигов). Для последовательного сложения этих n чисел

нужно

$$\underbrace{O(n) + O(n) + \ldots + O(n)}_{n-1 \text{ раз}} = O(n^2)$$

шагов. Другими словами, время работы *квадратично* относительно размера входов — по-прежнему полиномиально, хотя и гораздо больше, чем для сложения (где время линейно).

Аль-Хорезми знал другой способ умножения чисел. Запишем множители x и y рядом (см. пример ниже). Теперь будем повторять следующую операцию: поделим первое число пополам, отбросив дробную часть $1/2$ (если число было нечётным), а второе число удвоим. Будем делать так до тех пор, пока первое число не станет единицей. После этого вычеркнем все строки, в которых первое число чётно, и сложим оставшиеся числа из второй колонки.

11	13	
5	26	
2	52	(вычёркиваем)
1	104	
	143	(ответ)

На самом деле два этих алгоритма умножения (столбиком в двоичной системе и по способу аль-Хорезми) тесно связаны. Глядите: числа 13, 26 и 104, которые складываются в примере, получаются умножением числа 13 на степени двойки и в точности совпадают с промежуточными результатами при умножении столбиком в двоичной системе. Разница лишь в том, что во втором примере мы пользуемся десятичной записью, и поэтому двоичное представление 11 приходится строить, глядя на чётность чисел в левой колонке. Так что алгоритм Аль-Хорезми неявно использует двоичную систему, хотя все числа записываются в десятичной.

Можно даже сказать, что мы записали по существу один и тот же алгоритм двумя способами. Для разнообразия мы приведём и третью формулировку — рекурсивный алгоритм Multiply (рис. 1.1), который реализует простое правило:

$$x \cdot y = \begin{cases} 2\left(x \cdot \left\lfloor \frac{y}{2} \right\rfloor\right), & \text{если } y \text{ чётно,} \\ x + 2\left(x \cdot \left\lfloor \frac{y}{2} \right\rfloor\right), & \text{если } y \text{ нечётно.} \end{cases}$$

Корректен ли данный алгоритм? Да, поскольку он следует этому правилу при $y > 0$ и правильно ведёт себя в случае $y = 0$.

Каково время работы алгоритма? Если множитель y содержит n битов, то потребуется n рекурсивных вызовов, поскольку при каждом таком вызове y делится пополам (отбрасывается последний бит) и количество битов в двоичной записи y уменьшается на 1.

В каждом рекурсивном вызове производятся следующие операции: деление пополам (правый сдвиг), проверка на чётность (чтение последнего бита), умножение на 2 (левый сдвиг) и, возможно, сложение — всего $O(n)$ бито-

Рис. 1.1. Алгоритм умножения.

```
функция MULTIPLY(x, y)
{Вход: множители x и y, где y ⩾ 0.}
{Выход: xy.}
если y = 0:  вернуть 0
z ← MULTIPLY(x, ⌊y/2⌋)
если y чётно:
  вернуть 2z
иначе:
  вернуть x + 2z
```

вых операций для n-битовых входов. Общее время работы алгоритма, таким образом, будет по-прежнему $O(n^2)$.

Можно ли быстрее? Интуитивно кажется, что умножение требует n сложений, каждое из которых требует $O(n)$ шагов, поэтому всего нужно не меньше $\Omega(n^2)$ операций. Как ни странно, это не так, более быстрый алгоритм существует, и мы рассмотрим его в главе 2.

Нам осталось научиться делить числа. Поделить целое число x на положительное целое число y означает найти такие числа q (частное) и r (остаток), что $x = yq + r$ и $0 \leqslant r < y$. В упражнении 1.8 читателю предлагается доказать, что рекурсивный алгоритм DIVIDE (рис. 1.2) корректен и имеет квадратичное время работы.

Рис. 1.2. Алгоритм деления.

```
функция DIVIDE(x, y)
{Вход: n-битовые x и y, причём y ⩾ 1.}
{Выход: частное и остаток от деления x на y.}
если x = 0:  вернуть (q, r) = (0, 0)
(q, r) ← DIVIDE(⌊x/2⌋, y)
q ← 2·q, r ← 2·r
если x нечётно:  r ← r + 1
если r ⩾ y:  r ← r − y, q ← q + 1
вернуть (q, r)
```

1.2. Арифметика сравнений

При сложении и особенно умножении размеры чисел могут сильно возрасти. Иногда полезно «сбрасывать» ставшие слишком большими числа. Скажем, мы сбрасываем количество часов на ноль, когда оно достигает 24, а «на смену декабрям // приходят январи». Похожим образом числа в процессоре имеют некоторую максимальную длину, например, 32 бита. Подразумевается, что

обычно мы не выходим за этот предел, но когда выходим, процессор обрезает результат (оставляя последние 32 бита).

В интересующих нас приложениях (проверка числа на простоту, криптография) используются большие числа (гораздо длиннее 32 битов), но и они имеют ограниченную длину, и при сложении и умножении используется *арифметика остатков*, или *сравнений*. Сейчас мы объясним, что это такое.

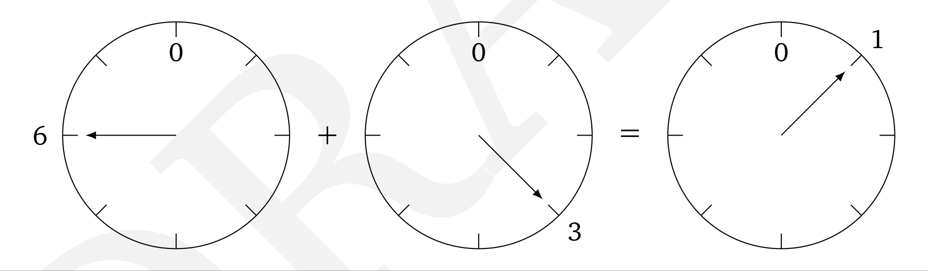
Фиксируем некоторое целое положительное N. Определим $x \bmod N$ (*x по модулю N*) как остаток от деления x на N: если $x = qN + r$, где $0 \leqslant r < N$, то $x \bmod N = r$. Это позволяет ввести отношение эквивалентности на числах: *x сравнимо с y по модулю N*, если x и y дают при делении на N одинаковый остаток, то есть если их разность делится на N:

$$x \equiv y \pmod N \quad \Longleftrightarrow \quad (x - y) \text{ делится на } N.$$

Например, $253 \equiv 13 \pmod{60}$, потому что $253 - 13 = 240$ делится на 60. (В более знакомой формулировке: 253 минуты — это 4 часа и 13 минут.) Сравниваемые числа могут быть и отрицательными: $59 \equiv -1 \pmod{60}$, ведь 59 минут — это час без одной минуты.

Работая с числами по модулю N, мы ограничиваемся отрезком $\{0, 1, \ldots \ldots, N-1\}$, и, достигнув конца отрезка, мы просто попадаем в его начало — как на циферблате (рис. 1.3).

Рис. 1.3. Сложение по модулю 8.



Можно также представлять себе, что мы работаем со всеми целыми числами, но они поделены на *N классов эквивалентности*, каждый из которых представляет собой арифметическую прогрессию с разностью N, то есть множество $\{i + kN : k \in \mathbb{Z}\}$ для некоторого i из интервала $0, \ldots, N-1$. Например, есть три класса эквивалентности по модулю 3:

$$\begin{array}{ccccccccc} \ldots & -9 & -6 & -3 & 0 & 3 & 6 & 9 & \ldots \\ \ldots & -8 & -5 & -2 & 1 & 4 & 7 & 10 & \ldots \\ \ldots & -7 & -4 & -1 & 2 & 5 & 8 & 11 & \ldots \end{array}$$

Работая по модулю 3, мы можем не различать числа одного класса, свободно заменяя одно на другое. Сложение и умножение согласованы с такими заменами (см. упражнение 1.9):

Корректность операций. *Если* $x \equiv x' \pmod N$ *и* $y \equiv y' \pmod N$*, то*

$$x + y \equiv x' + y' \pmod N \quad \text{и} \quad xy \equiv x'y' \pmod N$$

Для примера рассмотрим следующий вопрос: допустим, вы смотрите свой любимый сериал серию за серией, начав в полночь (0 часов); всего в нём 25 эпизодов, каждый из которых занимает три часа; в котором часу он кончится? Ответ вычислить легко: чтобы найти $(25 \times 3) \bmod 24$, заменим 25 на 1 (поскольку $25 \equiv 1 \pmod{24}$) и получим $1 \times 3 = 3 \mod 24$, то есть три часа ночи.

Легко убедиться, что сложение и умножение по модулю N по-прежнему обладают свойствами коммутативности, ассоциативности и дистрибутивности:

$$
\begin{aligned}
x + (y + z) &\equiv (x + y) + z \pmod{N} && \text{ассоциативность} \\
xy &\equiv yx \pmod{N} && \text{коммутативность} \\
x(y + z) &\equiv xy + yz \pmod{N} && \text{дистрибутивность}
\end{aligned}
$$

(подразумевается, что все операции выполняются по модулю N).

Можно сказать ещё и так: при выполнении арифметических операций по модулю N любые промежуточные результаты можно заменять на их остатки по модулю N. Это часто упрощает дело: к примеру,

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}.$$

1.2.1. Сложение и умножение по модулю

Чтобы *сложить* два числа x и y по модулю N, мы должны их сложить обычным образом и затем взять остаток при делении на N. Поскольку x и y лежат между 0 и $N-1$, их сумма лежит в интервале от 0 до $2(N-1)$. Если сумма превосходит $N-1$, нам остаётся лишь вычесть N. Таким образом, всё вычисление состоит из сложения, сравнения и (возможно) вычитания целых чисел, не превосходящих $2N$. Время работы будет линейно, то есть $O(n)$, где $n = \lceil \log N \rceil$ есть размер двоичной записи числа N.

Умножение чисел x и y по модулю N делается точно так же: сперва мы перемножаем числа, после чего берём остаток по модулю N. Произведение не превосходит $(N-1)^2$ и занимает не более $2n$ битов, поскольку $\log[(N-1)^2] = 2\log(N-1) \leqslant 2n$. Чтобы найти остаток по модулю N, мы используем алгоритм деления. И умножение, и деление требуют квадратичного времени, и мы получаем квадратичный алгоритм.

Деление по модулю N (решение уравнения $ax \equiv b \pmod{N}$) является более сложной операцией. В отличие от обычной арифметики, где проблемы возникают только при делении на ноль, в арифметике сравнений есть и другие сложные случаи, которые мы рассмотрим ближе к концу главы. Мы увидим, что деление (если оно вообще возможно) может быть выполнено за кубическое время, то есть $O(n^3)$.

1.2.2. Возведение в степень по модулю N

В криптосистеме, которую мы рассмотрим чуть позже, нам придётся вычислять $x^y \bmod N$ для x, y и N, состоящих из нескольких сотен битов. Как это сделать быстро?

Дополнительный код

Хорошей иллюстрацией арифметики сравнений служит *дополнительный код*, часто используемый для представления целых чисел в компьютере. При этом используется n битов для представления чисел из интервала $[-2^{n-1}, 2^{n-1}-1]$:

- Неотрицательные целые из интервала от 0 до $2^{n-1}-1$ хранятся в обычном двоичном представлении (с нулём в качестве старшего бита).
- Для хранения отрицательных чисел $-x$, где $1 \leqslant x \leqslant 2^{n-1}$, берётся двоичное представление числа x, все его биты инвертируются и к нему прибавляется единица.

Этот странный способ станет гораздо понятнее, если сказать, что числа из интервала от -2^{n-1} до $2^{n-1}-1$ хранятся по модулю 2^n. При этом отрицательное число $-x$ представляется как $2^n - x$. В таком представлении сложение, вычитание и умножение можно выполнять просто по модулю 2^n, не беспокоясь о переполнениях (оставляя только последние n битов).

Результатом такого вычисления является остаток по модулю N, то есть число длиной несколько сотен битов. Однако само x^y может иметь гораздо большую длину. Даже если в двоичной записи x и y всего 20 битов, число x^y может быть равно, например, $(2^{19})^{(2^{19})} = 2^{19\cdot 524288}$ (в двоичной записи около десяти миллионов битов). Представьте себе, что будет, если y будет длины 500.

Поэтому нельзя делить на N в конце: мы должны производить все текущие вычисления по модулю N. Значение $x^y \bmod N$ можно вычислить, начав с единицы и y раз умножив на x (каждый раз по модулю N). При этом все промежуточные результаты

$$x \bmod N \to x^2 \bmod N \to x^3 \bmod N \to \ldots \to x^y \bmod N$$

не превосходят N, и каждое умножение будет сравнительно быстрым. Проблема в другом: скажем, если y содержит 500 битов, то нам потребуется $y-1 \approx 2^{500}$ умножений! Время работы такого алгоритма будет расти пропорционально y, то есть экспоненциально от *размера* y.

К счастью, можно возводить в степень быстрее: начав с x, будем последовательно *возводить в квадрат* по модулю N:

$$x \bmod N \to x^2 \bmod N \to x^4 \bmod N \to x^8 \bmod N \to \ldots \to x^{2^{\lfloor \log y \rfloor}} \bmod N.$$

Каждое умножение требует времени $O(\log^2 N)$, всего же умножений будет $\log y$. Чтобы вычислить $x^y \bmod N$, мы просто перемножаем те степени x, которые соответствуют ненулевым позициям в двоичной записи y. Например,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1.$$

Мы получили, таким образом, полиномиальный (от размера входа) по времени алгоритм.

Рис. 1.4. Алгоритм возведения в степень по модулю.

```
функция ModExp(x, y, N)
{Вход: n-битовые числа x, y ⩾ 0, N > 0.}
{Выход: x^y mod N.}
если y = 0: вернуть 1
z ← ModExp(x, ⌊y/2⌋, N)
если y чётно:
  вернуть z^2 mod N
иначе:
  вернуть x · z^2 mod N
```

Алгоритм ModExp (рис. 1.4) реализует ту же идею (возведение в квадрат) немного другим способом: он использует то, что

$$x^y = \begin{cases} (x^{\lfloor y/2 \rfloor})^2, & \text{если } y \text{ чётно,} \\ x \cdot (x^{\lfloor y/2 \rfloor})^2, & \text{если } y \text{ нечётно.} \end{cases}$$

Данный алгоритм очень похож на рассмотренный нами алгоритм умножения Multiply (рис. 1.1), где произведение $x \cdot 25$ вычислялось как сумма $x \cdot 16 + x \cdot 8 + x \cdot 1$. Разница только в том, что алгоритм умножения *складывает* числа вида $x \cdot 2^i$, получаемые *удваиванием* исходного числа x, а теперь мы *перемножаем* числа x^{2^i}, получаемые последовательным *возведением в квадрат*.

Обозначим через n максимальную длину чисел x, y, N в двоичной записи. Подобно алгоритму умножения, алгоритм ModExp делает не более n рекурсивных вызовов, на каждом из которых умножаются числа длины не более n (все операции производятся по модулю N и требуют времени $O(n^2)$). Значит, общее время работы есть $O(n^3)$.

1.2.3. Алгоритм Евклида

Перейдём к алгоритму, который был придуман Евклидом в Древней Греции (более двух тысяч лет назад). Он находит *наибольший общий делитель* (НОД; greatest common divisor, gcd) двух чисел a и b, то есть максимальное натуральное число, на которое делятся одновременно a и b.

В принципе можно было бы сначала разложить a и b на простые множители, после чего выделить общие. Например, $1035 = 3^2 \cdot 5 \cdot 23$ и $759 = 3 \cdot 11 \cdot 23$, поэтому их наибольший общий делитель есть $3 \cdot 23 = 69$. У нас, однако, нет эффективного алгоритма для разложения чисел на множители. Можно ли найти наибольший делитель как-нибудь иначе?

Евклид показал, что достаточно воспользоваться простым правилом:

Правило Евклида. *Для любых целых чисел $x \geqslant y \geqslant 0$ выполняется равенство*

$$\text{НОД}(x, y) = \text{НОД}(x \bmod y, y).$$

Доказательство. Мы докажем, что НОД$(x, y) =$ НОД$(x - y, y)$, после чего требуемое равенство получается последовательным вычитанием y из x.

Ясно, что любое число, которое делит оба числа x и y, делит также и $x - y$, поэтому НОД$(x, y) \leqslant$ НОД$(x - y, y)$. Аналогично, любое число, которое делит оба числа $x - y$ и y, делит также и их сумму x, поэтому НОД$(x, y) \geqslant$ $\geqslant$ НОД$(x - y, y)$. □

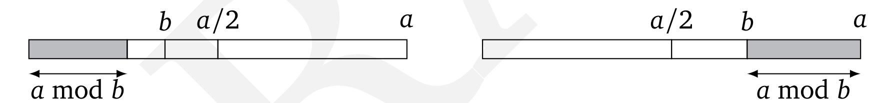
Правило Евклида позволяет записать элегантный рекурсивный алгоритм (рис. 1.5), корректность которого теперь очевидна. Для оценки времени работы нам нужно понять, как быстро уменьшаются значения аргументов. От пары (a, b) алгоритм переходит к паре $(b, a \bmod b)$, то есть большее из чисел (a) заменяется на $a \bmod b$.

Рис. 1.5. Алгоритм Евклида для вычисления наибольшего общего делителя.

функция Euclid(a, b)
{Вход: целые числа $a \geqslant b \geqslant 0$.}
{Выход: НОД(a, b).}
если $b = 0$: вернуть a
вернуть Euclid$(b, a \bmod b)$

Лемма. *Если $a \geqslant b$, то $a \bmod b < a/2$.*

Доказательство. Если $b \leqslant a/2$, то $a \bmod b < b \leqslant a/2$. Если же $b > a/2$, то $a \bmod b = a - b < a/2$. Оба случая показаны на рисунке ниже.



Лемма доказана. □

Данная лемма гарантирует, что после двух последовательных рекурсивных вызовов оба числа a и b уменьшатся хотя бы вдвое. Другими словами, длина каждого из них уменьшится хотя бы на один бит. Таким образом, будет произведено не более $2n$ рекурсивных вызовов. Поскольку на каждом из них производится деление, требующее квадратичного времени, общее время работы будет $O(n^3)$.

1.2.4. Расширенный алгоритм Евклида

Допустим, что кто-то утверждает, что нашёл наибольший общий делитель d чисел a и b. Как он может убедить нас в том, что это действительно так? Проверить, что d является общим делителем a и b, недостаточно — вдруг он не наибольший? Оказывается, что в качестве подтверждения достаточно предъявить представление d в виде суммы вида $ax + by$:

Лемма. *Если d делит оба числа a и b, а также $d = ax + by$ для некоторых целых чисел x и y, то $d =$ НОД(a, b).*

Доказательство. Поскольку d является общим делителем a и b, то $d \leqslant \leqslant$ НОД(a, b). С другой стороны, раз НОД(a, b) делит a и b, он должен делить также и $ax + by = d$. Следовательно, $d \geqslant$ НОД(a, b). □

Но всегда ли можно найти такое подтверждение, то есть коэффициенты x и y? Оказывается, что *всегда*, воспользовавшись *расширенным алгоритмом Евклида* (ExtendedEuclid, рис. 1.6).

Рис. 1.6. Расширенный алгоритм Евклида.

```
функция ExtendedEuclid(a, b)
{Вход: целые числа a ⩾ b ⩾ 0.}
{Выход: целые числа x, y, d, для которых d = НОД(a, b) и ax + by = d.}
если b = 0:   вернуть (1, 0, a)
(x′, y′, d) ← ExtendedEuclid(b, a mod b)
вернуть (y′, x′ − ⌊a/b⌋y′, d)
```

Лемма. *Для произвольных неотрицательных чисел a и b ($a \geqslant b$) расширенный алгоритм Евклида возвращает целые числа x, y, d, для которых* НОД$(a, b) = = d = ax + by$.

Доказательство. Легко видеть, что если выкинуть из алгоритма x и y, то получится просто алгоритм Евклида, поэтому он действительно вычисляет $d =$ НОД(a, b).

Оставшееся утверждение будем доказывать индукцией по b. База ($b = 0$) очевидна. Шаг индукции: заметим, что алгоритм находит НОД(a, b), произведя рекурсивный вызов для $(b, a \bmod b)$. Поскольку $a \bmod b < b$, мы можем воспользоваться предположением индукции и заключить, что для возвращаемых рекурсивным вызовом чисел x' и y' выполняется равенство

$$\text{НОД}(b, a \bmod b) = bx' + (a \bmod b)y'.$$

Переписав $(a \bmod b)$ как $(a - \lfloor a/b \rfloor b)$, получаем:

$$d = \text{НОД}(a, b) = \text{НОД}(b, a \bmod b) = bx' + (a \bmod b)y' = \\ = bx' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right) y' = ay' + b\left(x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right).$$

Таким образом, $d = ax + by$ при $x = y'$ и $y = x' - \lfloor a/b \rfloor y'$, что завершает доказательство корректности алгоритма. □

Пример. Алгоритм Евклида вычисляет НОД(25, 11) следующим образом:

$$\underline{25} = 2 \cdot \underline{11} + 3$$
$$\underline{11} = 3 \cdot \underline{3} + 2$$
$$\underline{3} = 1 \cdot \underline{2} + 1$$
$$\underline{2} = 2 \cdot \underline{1} + 0$$

(на каждой итерации вызов алгоритма происходит для подчёркнутых чисел). Таким образом, НОД(25, 11) = НОД(11, 3) = НОД(3, 2) = НОД(2, 1) = = НОД(1, 0) = 1.

Для нахождения x и y, для которых $25x + 11y = 1$, мы должны сначала представить 1 как линейную комбинацию чисел последней пары (1, 0). После этого мы возвращаемся и представляем его как линейную комбинацию чисел в парах (2, 1), (3, 2), (11, 3) и (25, 11). Первый шаг:

$$1 = \underline{1} - \underline{0}.$$

Чтобы выразить 1 через числа пары (2, 1), мы заменяем 0 на его выражение из предыдущего шага: $0 = 2 - 2 \cdot 1$, получается

$$1 = \underline{1} - (\underline{2} - 2 \cdot \underline{1}) = -1 \cdot \underline{2} + 3 \cdot \underline{1}.$$

Далее вспоминаем, что $1 = 3 - 1 \cdot 2$, и продолжаем:

$$1 = -1 \cdot \underline{2} + 3 \cdot \underline{1} = -1 \cdot \underline{2} + 3 \cdot (\underline{3} - 1 \cdot \underline{2}) = 3 \cdot \underline{3} - 4 \cdot \underline{2}.$$

Поскольку $2 = \underline{11} - 3 \cdot \underline{3}$ и затем $3 = \underline{25} - 2 \cdot \underline{11}$, то

$$1 = 3 \cdot \underline{3} - 4 \cdot (\underline{11} - 3 \cdot \underline{3}) = -4 \cdot \underline{11} + 15 \cdot \underline{3} = \\ = -4 \cdot \underline{11} + 15 \cdot (\underline{25} - 2 \cdot \underline{11}) = 15 \cdot \underline{25} - 34 \cdot \underline{11}.$$

1.2.5. Деление по модулю

В обычной арифметике у каждого ненулевого числа a есть обратное $1/a$, и разделить на a — то же самое, что домножить на $1/a$. Можно дать аналогичное определение и в арифметике остатков:

> Будем называть число x *мультипликативно обратным* (multiplicative inverse) к a по модулю N, если $ax \equiv 1 \pmod{N}$.

Можно доказать, что такое x единственно (по модулю N, упражнение 1.23), поэтому можно ввести для него обозначение a^{-1}. Однако существует такое число не всегда. Например, число 2 не обратимо по модулю 6, поскольку число $2x$ чётно и не может давать остаток 1 при делении на чётное число 6.

Вообще, если a и N имеют общий делитель, то есть $d = \text{НОД}(a, N) > 1$, то a не имеет обратного по модулю N. В самом деле, если $ax \equiv 1 \pmod{N}$, то $ax - 1$ делится на N, то есть $ax - 1 = kN$ при некотором k. Тогда $1 = ax - kN$, и получается противоречие, так как левая часть не делится на d, а оба числа в правой части делятся.

На самом деле, это — единственная ситуация, в которой a не обратимо по модулю N. Если $\text{НОД}(a, N) = 1$ (в таком случае мы называем числа a и N *взаимно простыми*), расширенный алгоритм Евклида найдёт такие x и y, что $ax + Ny = 1$, из чего следует, что $ax \equiv 1 \pmod{N}$. Число x, таким образом, будет обратным к a по модулю N.

Пример. Продолжая наш предыдущий пример, допустим, что мы хотим вычислить $11^{-1} \bmod 25$. Используя расширенный алгоритм Евклида, находим,

что $15 \cdot 25 - 34 \cdot 11 = 1$. Переходя к модулю 25, получаем $-34 \cdot 11 \equiv 1 \pmod{25}$. Поэтому $-34 = 16 \bmod 25$ является обратным к 11 по модулю 25.

Обращение по модулю N. *У числа a есть обратное по модулю N тогда и только тогда, когда a и N взаимно просты. Когда обратное существует, оно может быть найдено за время $O(n^3)$ (где n, как обычно, обозначает размер входных данных) расширенным алгоритмом Евклида.*

Итак, с делением по модулю мы разобрались: в арифметике по модулю N можно делить на числа, взаимно простые с N. Деление на такое число состоит в умножении на обратное к нему, которое можно найти с помощью расширенного алгоритма Евклида.

1.3. Проверка чисел на простоту

Как же выяснить, является ли число простым, не пытаясь раскладывать его на множители? В этом нам поможет теорема 1640 года:

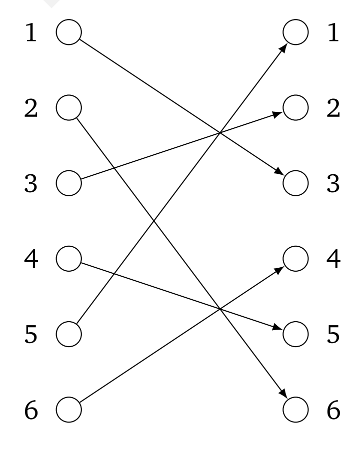
Малая теорема Ферма. *Для любого простого числа p и любого $1 \leqslant a \leqslant p-1$ выполнено сравнение*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Доказательство. Пусть $S = \{1, 2, \ldots, p-1\}$ — множество ненулевых остатков по модулю p. Важное наблюдение: умножая все эти числа на a (по модулю p), мы опять получим то же самое множество. На рисунке ниже показан пример для $a = 3$, $p = 7$. Из рисунка видно, что числа

$$3 \cdot 1 \bmod 7, \quad 3 \cdot 2 \bmod 7, \quad \ldots, \quad 3 \cdot 6 \bmod 7$$

представляют собой числа 1, 2, …, 6, только записанные в другом порядке. Перемножив их все, получаем, что $6! \equiv 3^6 \cdot 6! \pmod{7}$. Сокращая на $6!$ (это допустимо, так как числа 1, 2, …, 6 взаимно просты с 7, и можно по очереди сократить на каждое из них), находим, что $3^6 \equiv 1 \pmod{7}$ — как раз то, что требовалось доказать в случае $a = 3$, $p = 7$.



Является ли число простым?

Числа 7, 17, 19, 71, 79 являются простыми, но как насчёт числа 717197179? С первого взгляда вопрос непростой, поскольку кандидатов на роль его делителей много. Есть, однако, разные способы ограничить перебор. Например, если число нечётно, то не стоит проверять, есть ли у него чётные делители. Более того, в качестве кандидатов можно рассматривать только простые числа.

Далее, число N является простым, если у него нет делителей от 2 до $\sqrt{N}$. Действительно, в разложении $N = K \cdot L$ хотя бы одно из чисел K и L обязано быть не больше $\sqrt{N}$.

Как видно, немного подумав, мы смогли довольно сильно сократить перебор. Быть может, продолжая в том же духе, можно получить эффективный алгоритм проверки числа на простоту? Пока это никому не удалось. Дело в том, что мы пытались проверить число на простоту, одновременно *разложив его на множители*, а быстро разлагать числа на множители пока никто не умеет.

Современная криптография основана на следующей важной идее: *раскладывать число на множители трудно, в то время как проверить число на простоту легко*. Оказывается, что можно быстро проверить, является ли данное число простым — но проверка эта не даёт делителей, когда оно оказывается составным!

Аналогичное рассуждение годится и для произвольного простого p (и произвольного a). Первым делом мы покажем, что, умножая (по модулю p) элементы множества S на p, мы получим *различные* ненулевые остатки по модулю p. Отсюда будет следовать, что встречаются *все* ненулевые остатки (их столько же).

Числа $a \cdot i \bmod p$ различны при разных i, поскольку из $a \cdot i \equiv a \cdot j \pmod p$ делением на a (умножением на a^{-1}) получается, что $i \equiv j \pmod p$. (Вспомним, что число p простое, поэтому a взаимно просто с p, и a^{-1} существует.)

Все $a \cdot i \bmod p$ не равны нулю, поскольку (опять же) из $a \cdot i \equiv 0 \pmod p$ следует, что $i \equiv 0 \pmod p$ (деление на a).

Теперь можно записать S двумя разными способами:

$$S = \{1, 2, \ldots, p-1\} = \{a \cdot 1 \bmod p, a \cdot 2 \bmod p, \ldots, a \cdot (p-1) \bmod p\}.$$

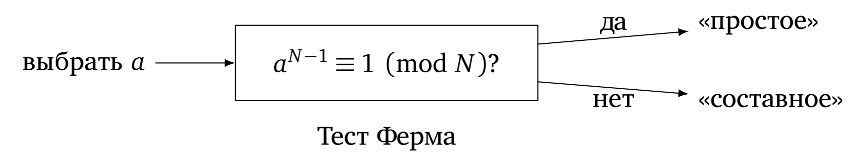
Произведение от порядка не зависит, так что

$$(p-1)! \equiv a^{p-1} \cdot (p-1)! \pmod p.$$

Сократив на $(p-1)!$ (последовательно умножая на обратные к $2, 3, \ldots, p-1$, которые существуют, так как $2, 3, \ldots, p-1$ взаимно просты с p), получаем утверждение теоремы. □

Эта теорема позволяет установить, что число p составное, не находя его делителей: достаточно убедиться, что $a^{p-1} \not\equiv 1 \pmod p$ при некотором $a = 1, 2, \ldots, p-1$. Может быть, можно проверять простоту чисел с её помощью?

Например, так:



Тест Ферма

Проблема в том, что условие в теореме Ферма не является необходимым (хотя является достаточным): теорема ничего не говорит про случай, когда N составное. И действительно, составные числа N вполне могут проходить тест Ферма (то есть удовлетворять сравнению $a^{N-1} \equiv 1 \pmod{N}$) для некоторых значений a. Например, число $341 = 31 \cdot 11$ составное, но в то же время $2^{340} \equiv 1 \pmod{341}$.

Можно всё же надеяться, что для составных N таких значений *a не очень много*. В каком-то смысле это действительно так (точную формулировку мы дадим позже), и на этой идее основан алгоритм Primality (рис. 1.7), который использует не какое-то фиксированное a, а выбирает его *случайным образом* из множества $\{1, 2, \ldots, N-1\}$.

Рис. 1.7. Алгоритм проверки числа на простоту.

```
функция Primality(N)
{Вход: положительное целое число N.}
{Выход: да/нет.}
взять случайное целое число a от 1 до N − 1
если a^(N−1) ≡ 1 (mod N):
  вернуть «да»
иначе:
  вернуть «нет»
```

Что можно сказать про этот алгоритм? Плохая новость: существуют составные числа N, которые проходят тест Ферма для всех a (взаимно простых с N). Эти числа наш алгоритм сочтёт простыми. Их называют *числами Кармайкла* (Carmichael numbers).

Хорошая новость: во-первых, числа Кармайкла встречаются довольно редко; во-вторых, как мы скоро увидим (с. 32), с ними можно справиться другим способом.

На остальных числах (кроме чисел Кармайкла) наш алгоритм работает довольно хорошо. В самом деле, любое простое число, конечно же, пройдёт тест и алгоритм выдаст правильный ответ. А что будет с составными числами? По определению, составное число, не являющееся числом Кармайкла, не проходит тест хотя бы при одном значении a. Мы покажем, что такие числа не проходят тест *для половины всех возможных значений a* (или даже больше).

Лемма. *Если $a^{N-1} \not\equiv 1 \pmod{N}$ для некоторого a, взаимно простого с N, то это верно не менее чем для половины всех a в интервале от 1 до $N-1$.*

Это же теория групп!

Для любого положительного целого числа N множество чисел от 1 до N, взаимно простых с N, образует *группу* (group). Это значит, что:

- на множестве определена операция умножения;
- множество содержит нейтральный элемент (единицу): при умножении на него любой элемент остаётся неизменным;
- у каждого элемента есть обратный (который в произведении даёт единицу).

Рассматриваемая нами группа называется *мультипликативной группой вычетов по модулю* N (multiplicative group of N) и обозначается $\mathbb{Z}_N^*$.

Теория групп — важный раздел математики. Одним из ключевых понятий в ней является понятие *подгруппы* (subgroup). Это подмножество группы, само являющееся группой (с той же операцией умножения). Теорема Лагранжа, один из первых результатов теории групп, утверждает, что размер подгруппы (число элементов в ней) делит размер группы.

Рассмотрим теперь множество $B = \{b : b^{N-1} \equiv 1 \pmod N\}$. Нетрудно показать, что это подгруппа $\mathbb{Z}_N^*$ (нужно просто проверить, что B замкнуто относительно умножения и взятия обратного). Значит, размер B обязан делить размер $\mathbb{Z}_N^*$. И если B не совпадает с $\mathbb{Z}_N^*$, то его максимальный размер есть $|\mathbb{Z}_N^*|/2$ и заведомо не больше $(N-1)/2$.

Доказательство. Зафиксируем a, взаимно простое с N, для которого $a^{N-1} \not\equiv 1 \pmod N$. Теперь заметим, что каждому остатку $b < N$, для которого тест проходит (то есть $b^{N-1} \equiv 1 \pmod N$), можно сопоставить остаток $a \cdot b$, для которого тест не проходит:

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1 \pmod N.$$

Более того, разным b соответствуют разные $a \cdot b$ (всё по той же причине: раз a обратимо, то $a \cdot i \not\equiv a \cdot j$ для $i \not\equiv j$). Таким образом, остатков, для которых тест не проходит, не меньше, чем тех, для которых он проходит. □

Таким образом, для всех чисел N, кроме чисел Кармайкла, верны следующие утверждения.

Если N простое, то $a^{N-1} \equiv 1 \pmod N$ для всех $a < N$.

Если N составное, то $a^{N-1} \equiv 1 \pmod N$ для не более чем половины всех $a < N$.

Значит, для алгоритма Primality верны следующие оценки вероятности:

$$\text{если } N \text{ простое, то } P(\text{алгоритм возвращает «да»}) = 1;$$

$$\text{если } N \text{ составное, то } P(\text{алгоритм возвращает «да»}) \leqslant \frac{1}{2}.$$

В таких случаях говорят об *односторонней ошибке* (one-sided error): мы можем принять составное число за простое, но не наоборот. Вероятность

ошибки можно уменьшить, повторив тест для нескольких случайно выбранных a (алгоритм Primality2 на рис. 1.8): для составного N

$$\mathrm{P}(\text{алгоритм Primality2 возвращает «да»}) \leqslant \frac{1}{2^k}.$$

Вероятность ошибки быстро (экспоненциально) убывает с ростом k. Скажем, при $k = 100$ получается 2^{-100}, что уже пренебрежимо мало (скорее уж компьютер во время вычислений даст сбой).

Рис. 1.8. Алгоритм проверки простоты числа с низкой вероятностью ошибки.

функция Primality2(N)
{Вход: положительное целое число N.}
{Выход: да/нет.}
взять k случайных целых положительных чисел $a_1, a_2, \ldots, a_k < N$
если $a_i^{N-1} \equiv 1 \pmod{N}$ для всех $i = 1, 2, \ldots, k$:
  вернуть «да»
иначе:
  вернуть «нет»

1.3.1. Генерация случайных простых чисел

У нас уже почти всё готово для криптографических приложений. Последнее, чему нам осталось научиться, — это генерировать случайные простые числа длиной в несколько сотен битов. Это не очень сложно, поскольку простых чисел довольно много: случайное число из n битов имеет примерно один шанс из n быть простым (точнее, около $1/(\ln 2^n) \approx 1{,}44/n$).

Закон распределения простых чисел. *Обозначим через $\pi(x)$ количество простых чисел, не превосходящих x. Тогда $\pi(x) \approx x/(\ln x)$ или, более формально,*

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1.$$

Закон этот имеет долгую историю (начиная с Лежандра) и доказывается совсем не просто, но благодаря ему можно довольно быстро генерировать случайные простые числа: нужно всего лишь брать случайные числа требуемой длины и проверять их на простоту, пока не обнаружится простое.

Каково же ожидаемое время работы приведённого алгоритма? Сколько попыток в среднем нужно, чтобы наткнуться на простое число (которое гарантированно пройдёт тест)? Вероятность случайно наткнуться на простое число (для каждой попытки) не меньше $1/n$ (закон распределения простых чисел). Теория вероятностей позволяет заключить, что в среднем понадобится не более n попыток (упражнение 1.34).

Остаётся понять, как в этом алгоритме выполнять проверку на простоту и какова будет вероятность ошибки (того, что алгоритм выдаст составное

Числа Кармайкла

Наименьшее число Кармайкла равно 561. Оно является составным ($561 = 3 \cdot 11 \cdot 17$) и в то же время проходит тест Ферма для всех a, взаимно простых с 561 (то есть $a^{560} \equiv 1 \pmod{561}$), и таких a большинство.

Недавно было показано, что чисел Кармайкла бесконечно много.

Рабин и Миллер предложили более сложный способ вероятностной проверки на простоту, который годится и для чисел Кармайкла. Пусть нам дано нечётное N. В чётном числе $N-1$ выделим все двойки, то есть представим его в виде $N-1 = 2^t u$ с нечётным u. Как и раньше, выберем случайное a и вычислим $a^{N-1} \bmod N$. При этом мы сначала вычислим $a^u \bmod N$, а потом возведём его в квадрат t раз:

$$a^u \bmod N,\ a^{2u} \bmod N,\ \ldots,\ a^{2^t u} \equiv a^{N-1} \bmod N.$$

Если $a^{N-1} \not\equiv 1 \pmod{N}$, то N (как мы уже знаем) не является простым. В противном случае сделаем дополнительную проверку: посмотрим, когда в этой последовательности впервые появилась единица по модулю N, и посмотрим на предыдущий член последовательности. Если он есть (то есть последовательность не начинается с единицы) и не равен $-1 \bmod N$, то мы заключаем, что N составное. В самом деле, в этом случае мы нашли *нетривиальный корень из единицы* (nontrivial square root of 1) — число, которое не сравнимо с ± 1 по модулю N, но его квадрат сравним с 1. Такое число может существовать только в случае, если N составное (упражнение 1.40).

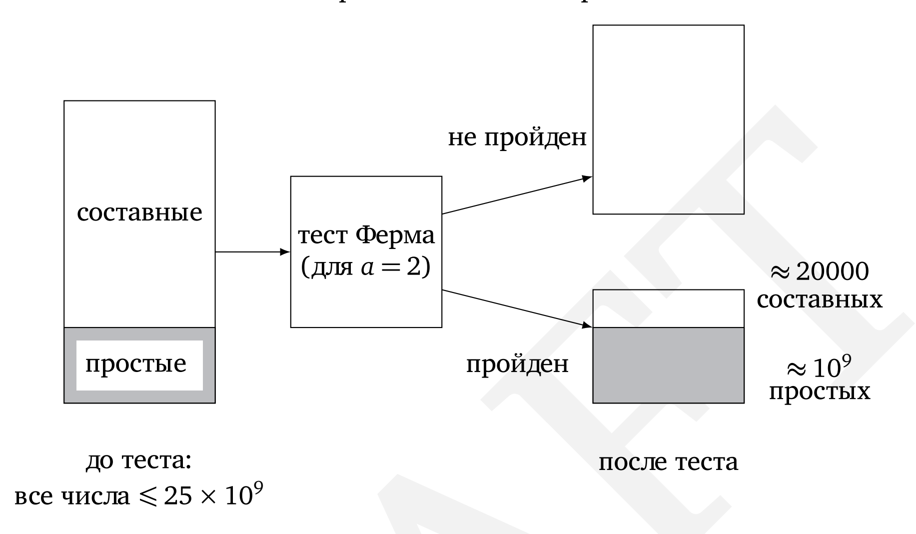
Оказывается, что дополненный таким образом тест универсален, и числа Кармайкла ему уже не страшны: при *любом* составном N с вероятностью $3/4$ или больше мы это обнаружим.

число). Похоже, что с практической точки зрения достаточно выполнять тест Ферма для $a = 2$ (для особо недоверчивых — при $a = 2, 3, 5$) и всё равно с большой вероятностью мы найдём простое число.

Почему? Из доказанных нами фактов это не следует, и сразу по двум причинам. Во-первых, существуют числа Кармайкла, где никакое a не поможет. Во-вторых, и для остальных чисел мы знаем лишь, что по меньшей мере половина всех a позволит выявить их непростоту, а про конкретное значение $a = 2$ ничего не знаем.

С другой стороны, численные эксперименты делают это предположение правдоподобным. Запустим тест Ферма для всех положительных целых чисел $N \leqslant 25 \times 10^9$ и $a = 2$. В этом диапазоне содержится около 10^9 простых чисел и все они, естественно, тест пройдут. Кроме того, обнаруживается примерно 20000 чисел, которые проходят тест, хотя являются составными. Наш алгоритм (выбирать случайное число, пока одно из них не пройдёт тест) с равной вероятностью может выдать любое из этих чисел (примерно 10^9 простых и примерно 20000 составных). Значит, шанс попасть на составное число примерно равен $2 \cdot 10^{-5} = 0{,}002\%$.

Можно надеяться, что для чисел большего размера эта вероятность будет ещё меньше. Выигрыш за счёт упрощения теста во многих ситуациях перевешивает опасность, и ей часто пренебрегают. Полученные таким образом числа можно было бы назвать «простыми числами промышленного качества».



Вероятностные алгоритмы: виртуальная глава

Удивительно, но некоторые из самых быстрых и изощрённых алгоритмов, придуманных для разных задач, являются *вероятностными*: алгоритм иногда подбрасывает монету и действует в зависимости от исхода. При этом вероятность успеха на любом входе (доля случаев, в которой *на этом входе* будет получен правильный ответ) оказывается близкой к единице. (Важно не перепутать их с алгоритмами, которые дают правильный ответ на большинстве входов: алгоритм, который говорит «составное» про любое 1000-битовое число, ошибается менее чем в 1% случаев, но пользы от него никакой.) Мы решили не выделять вероятностные алгоритмы в отдельную главу, а рассматривать их по ходу дела. При этом для их понимания, как правило, достаточно интуитивных представлений о базовых понятиях теории вероятностей (вероятность события, математическое ожидание и его линейность).

Перечислим основные вероятностные алгоритмы, встречающиеся в нашей книге. Мы уже рассмотрели вероятностную проверку простоты (рис. 1.8). Вероятностная структура данных, называемая хеш-таблицей и позволяющая хранить множество (с операциями добавления, удаления и проверки принадлежности), описана в разделе 1.5 этой главы. В главе 2 приводятся вероятностные алгоритмы сортировки и нахождения медианы. Вероятностный алгоритм нахождения минимального разреза описан во врезке на с. 137. Случайность используется также в эвристических алгоритмах, некоторые из которых описаны в разделе 9.3. Наконец, квантовый

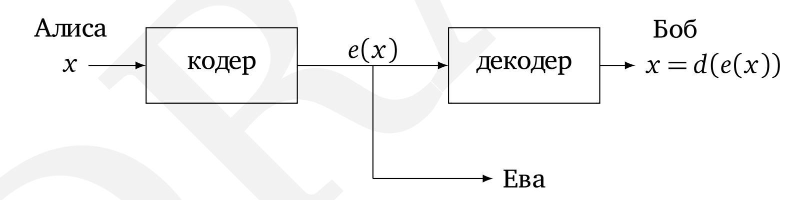
алгоритм разложения числа на множители (см. раздел 10.7) тоже в некотором смысле вероятностный, но, чтобы понять, что это значит, надо разобраться в основах квантовой механики.

Упражнения, в которых идёт речь о вероятностных алгоритмах: 1.29, 1.34, 2.24, 9.8, 10.8.

1.4. Криптография

Криптосистема RSA (RSA cryptosystem), названная в честь её создателей Р. Ривеста, А. Шамира и Л. Адлемана, которую мы рассмотрим в данном разделе, использует все идеи данной главы. Надёжность этой системы основана на предположении о том, что некоторые вычислительные задачи (возведение в степень по модулю, нахождение наибольшего общего делителя, проверка на простоту) решаются просто даже и для больши́х чисел, в то время как другие являются сложными и на практике невыполнимы (разложение больших чисел на множители).

Три любимых героя криптографов — Алиса (A) и Боб (B), которые хотят переговорить без свидетелей, и любопытная Ева (E), которая их подслушивает. (По-английски «подслушивающий» будет «eavesdropper», что объясняет выбор имени.) Пусть, скажем, Алиса хочет послать Бобу секретное сообщение — строку битов x. Она шифрует x с помощью функции $e(\cdot)$ и посылает Бобу зашифрованное сообщение $e(x)$. Боб использует функцию $d(\cdot)$, чтобы восстановить (дешифровать) исходное сообщение: $d(e(x)) = x$.



Алиса и Боб опасаются, как бы Ева не подслушала $e(x)$. Они надеются, что Ева, не знающая алгоритма расшифровки d, не сможет извлечь никакой информации об x из $e(x)$.

Довольно долго криптография была основана на использовании секретного («закрытого») ключа (private key). Таким ключом может быть, скажем, заранее составленная и известная только им двоим таблица для кодирования и декодирования (codebook). Не зная этой таблицы, Ева не сможет ничего понять (разве что одни и те же коды будут использоваться многократно, и это даст частичную информацию о сообщении).

Но как быть, если Алиса и Боб никогда не встречались и всю их переписку Ева может читать с самого начала? Смогут ли они передать друг другу что-то в секрете от неё? На первый взгляд кажется, что это невозможно: если из их переписки можно извлечь способ расшифровки, то он доступен Еве, если же нет — то как они сами смогут что-то расшифровать?

Оказывается, что это всё-таки возможно, если воспользоваться разницей между сложностью задачи проверки простоты (и отыскания простых чисел) и задачи разложения на множители. На этом основана система шифрования RSA с открытым ключом (Rivest—Shamir—Adleman public-key cryptosystem). При её использовании Боб сообщает Алисе некоторую информацию — «открытый ключ». С её использованием Алиса может зашифровать своё сообщение. Но расшифровать его сможет только Боб, поскольку одного открытого ключа для этого мало. (Эта схема используется при передаче секретных данных, типа номеров кредитных карт, по интернету.)

При этом Алисе и Бобу не нужно выполнять особо сложных вычислений для шифрования и дешифрования, с ними справится и мобильный телефон. Но расшифровка, хотя теоретически и возможна, требует умения раскладывать числа на множители, и даже суперкомпьютеры в этом не помогут. Именно за счёт этого разрыва система RSA произвела революцию в криптографии.

1.4.1. Схемы с закрытым ключом: одноразовый ключ и AES

В этом разделе мы обсудим два способа шифрования с закрытым ключом (они надёжны, только если противник не знает этого ключа).

В протоколе шифрования с *одноразовым ключом* (one-time pad) Алиса и Боб встречаются заранее и выбирают ключ — битовую строку r той же длины, что и будущее сообщение. Шифрование сообщения x состоит в побитовом сложении x с ключом r. Каждый бит строки $e(x)$ равен сумме по модулю два соответствующих битов x и r; эта операция называется XOR (eXclusive OR, «исключающее ИЛИ»): $e_r(x) = x \oplus r$. Например, если выбран ключ $r = 01110010$, то сообщение $x = 11110000$ кодируется как

$$e_r(11110000) = 11110000 \oplus 01110010 = 10000010.$$

Легко сообразить, что функция e_r обратна к самой себе (в том смысле, что, применив её второй раз, мы вернёмся к исходному сообщению):

$$e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus (r \oplus r) = x \oplus \bar{0} = x;$$

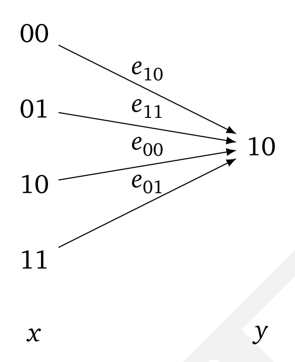
здесь $\bar{0}$ — строка из одних нулей. Так что дешифрование просто повторяет шифрование: $d_r(y) = y \oplus r$.

Как надо выбирать r, чтобы эта схема была надёжной? Простой способ: подбросить монетку столько раз, сколько битов в r, и записать результаты бросаний. При этом все варианты (все элементы множества $\{0, 1\}^n$) равнове-

И терпентин на что-нибудь полезен

Известный специалист по теории чисел Г. Харди однажды написал: «За свою жизнь я не сделал ничего „полезного“». Он был специалистом по теории чисел — самой что ни на есть «чистой» (не «прикладной») математике — и был бы очень удивлён, если бы ему рассказали, что теоремы теории чисел найдут самое непосредственное применение в банковских операциях.

роятны, и прибавление к ним любого сообщения x перемешивает их, оставляя равновероятными, так что Ева (не знающая r) ничего не узнает. Пусть, скажем, Ева видит сообщение 10. Это сообщение может получиться из любого сообщения Алисы при каком-то значении ключа, и никакой информации Ева не получает!



Конечно, эту схему можно использовать только однажды (отсюда и название). Если послать таким способом два сообщения x и z, то Ева сможет восстановить $x \oplus z$, поскольку $(x \oplus r) \oplus (z \oplus r) = (x \oplus z) \oplus (r \oplus r) = x \oplus z \oplus \bar{0} = x \oplus z$. Эта информация может быть существенной: скажем, по $x \oplus z$ видно, начинаются ли x и z на одно и то же. А если одно из сообщений содержит длинный кусок из подряд идущих нулей (что вполне возможно, если это картинка), то в $x \oplus z$ на соответствующем месте будет просто кусок другого сообщения. Поэтому «бывшие в употреблении» биты нельзя использовать повторно, и запаса общих случайных битов должно хватить на все будущие сообщения.

Система эта проста и надёжна, но непрактична (требует длинного ключа). Похожий (и в то же время широко использующийся) способ шифрования описан в *стандарте AES* (advanced encryption standard). Как и прежде, Алиса и Боб заранее выбирают случайную строку битов r, но фиксированной длины — 128 битов (есть варианты с длинами 192 и 256). Зная r, можно вычислять взаимно однозначную функцию e_r на строках длины 128. Ключевое отличие состоит в том, что эта функция используется много раз: длинное сообщение разбивают на куски длиной 128 битов и применяют e_r к каждому из них.

Надёжность стандарта AES не доказана. В то же время никто не знает (или, по крайней мере, не публикует!) способа расшифровки сообщений, не требующего перебора нереально большого числа вариантов.

1.4.2. Протокол RSA

В отличие от двух предыдущих, схема шифрования RSA является *протоколом с открытым ключом*. Если Боб хочет получать конфиденциальные сообщения, он публикует открытый ключ, с помощью которого любой желающий может шифровать посылаемые ему сообщения. А кроме этого, у Боба есть закрытый ключ, который нужен для расшифровки и который известен только

ему. Ева его не знает и потому расшифровать ничего не может (если только она не научилась быстро раскладывать числа на множители).

Схема RSA основана на теории чисел. Сообщения будем считать числами по модулю N (сообщения большей длины можно разбить на части). Шифрование будет перестановкой множества $\{0, 1, \ldots, N-1\}$, а дешифрование — обратной перестановкой. Осталось указать, как выбирать N и эти перестановки.

Свойство. *Пусть N — произведение двух простых чисел p и q, а e взаимно просто с $(p-1)(q-1)$. Тогда*:

1. *Отображение $x \mapsto x^e \bmod N$ является перестановкой остатков по модулю N (множества $\{0, 1, \ldots, N-1\}$).*

2. *Обратной перестановкой будет возведение в степень d, где d — обратное к e число по модулю $(p-1)(q-1)$:*

$$(x^e)^d \equiv x \pmod N$$

при всех $x = 0, 1, \ldots, N-1$.

Первое свойство гарантирует, что при шифровании $x \mapsto x^e$ информация не теряется. Опубликовав пару (N, e) как *открытый ключ*, Боб позволяет всем желающим выполнять операцию шифрования. Второе свойство позволяет быстро расшифровать сообщение, зная *закрытый ключ* d. (Зная только открытый ключ e, можно расшифровать сообщение перебором всех вариантов, но при большом N это долго.)

Пример. Пусть $N = 55 = 5 \cdot 11$. Возьмём $e = 3$; это значение e допустимо, так как НОД$(e, (p-1)(q-1)) =$ НОД$(3, 40) = 1$. В качестве d надо взять $3^{-1} \bmod 40 = 27$. Теперь кодом любого сообщения x будет $y = x^3 \bmod 55$, а декодирование выполняется так: $x = y^{27} \bmod 55$. Например, если $x = 13$, то $y = 13^3 \bmod 55 = 52$, а $13 = 52^{27} \bmod 55$.

Докажем теперь сформулированное утверждение и перейдём к анализу надёжности схемы.

Доказательство. Достаточно доказать второе утверждение: если у отображения есть обратное, то это перестановка. Прежде всего отметим, что e обратимо по модулю $(p-1)(q-1)$, поскольку e и $(p-1)(q-1)$ взаимно просты. Пусть $d = e^{-1} \bmod N$, то есть $ed \equiv 1 \pmod{(p-1)(q-1)}$, и $ed = 1 + k(p-1)(q-1)$ для некоторого k. Мы должны показать, что

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x$$

делится на N. По малой теореме Ферма $x^{p-1} \equiv 1 \pmod p$ при $x \not\equiv 0 \pmod p$, поэтому $x^{1+k(p-1)(q-1)} - x$ делится на p при всех x (в том числе и при $x \equiv 0 \pmod p$, в этом случае обе части равны 0 по модулю p). По той же причине $x^{ed} - x$ делится на q. Поскольку p и q взаимно просты, $x^{ed} - x$ делится и на произведение чисел p и q, то есть на N. □

Протокол RSA показан на рис. 1.9. Видно, что Алиса и Боб выполняют лишь простые действия. Но насколько протокол надёжен?

Рис. 1.9. Протокол RSA.

Боб выбирает открытый и закрытый ключи

- Сначала он выбирает два случайных простых числа p и q длиной n бит каждое.
- Открытый ключ Боба — это пара (N, e), где $N = pq$, а e — взаимно простое с $(p-1)(q-1)$ число. Часто в качестве e выбирают число 3, чтобы упростить кодирование.
- Закрытый ключ Боба — число d, обратное к e по модулю $(p-1)(q-1)$. Боб знает p и q и может найти d с помощью расширенного алгоритма Евклида.

Шифрование сообщения x

- Используя открытый ключ Боба (N, e), Алиса посылает Бобу число $y = x^e \bmod N$ (алгоритм возведения в степень по модулю мы разбирали).

Дешифрование

- Получив y, Боб вычисляет $x = y^d \bmod N$.

Надёжность протокола RSA основана на таком предположении:

Зная N, e и $y = x^e \bmod N$, нереально найти x.

Данное предположение выглядит правдоподобно (хотя не доказано). В самом деле, Ева могла бы перебирать все возможные x (и проверять равенство $x^e \equiv y \pmod N$), но это экспоненциально долго. Другой вариант: Ева могла бы разложить N на множители, после чего вычислить d (как это делает Боб), но разложение числа на множители кажется вычислительно трудной задачей.

Обратите внимание на парадоксальное обстоятельство: обычно для практики полезен быстрый алгоритм решения какой-то задачи; здесь же нам важно *отсутствие* быстрого алгоритма разложения на множители!

1.5. Универсальное хеширование

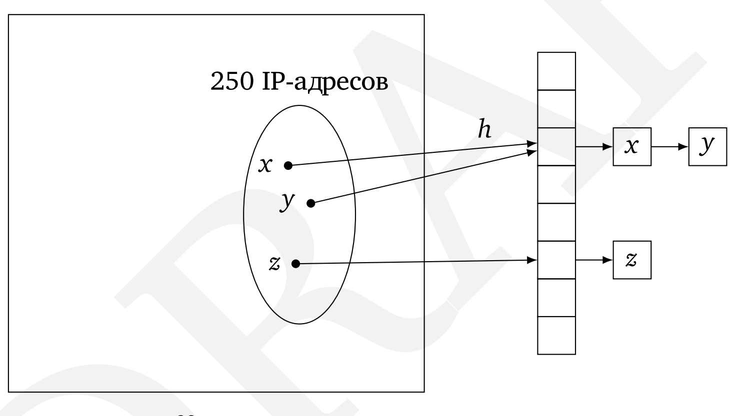
В этом разделе мы применим теорию чисел к построению *хеш-функций* (hash functions).

Представим себе веб-сервис, в котором нужно хранить информацию об активных клиентах, идентифицируемых по их IP-адресам, и таких клиентов, скажем, не более 250. (Напомним, что IP-адрес состоит из 32 битов и обычно записывается четырьмя числами от 0 до 255, разделённых точками, например, 18.3.7.144). Как быстро проверить, есть ли данный IP-адрес в списке?

Можно завести массив, в котором индексом будет сам IP-адрес, тогда проверка сведётся к просмотру одного элемента данного массива. Но такой массив должен состоять из $2^{32} \approx 4 \times 10^9$ ячеек, и почти все из них будут, как правило, пустыми. С другой стороны, можно хранить список всех адресов, но тогда поиск заданного адреса занимает время, пропорциональное длине списка. Хеширование позволяет совместить достоинства этих двух способов, избегая их недостатков.

1.5.1. Хеш-таблицы

Каждому из всех возможных 2^{32} IP-адресов поставим в соответствие «псевдоним». Будем считать, что псевдонимы — целые числа в диапазоне от 1 до 250 (позже мы слегка изменим этот диапазон). При этом, разумеется, многим IP-адресам соответствует одно и то же число. Мы, однако, будем надеяться, что активным адресам будут в основном соответствовать разные псевдонимы, и будем хранить эти адреса в массиве размера 250 (индекс — псевдоним).



Что делать в случаях, когда два активных адреса имеют один и тот же псевдоним? Будем хранить в массиве указатель на список, содержащий эти адреса. Объём используемой памяти при таком способе пропорционален количеству клиентов (250), а не количеству всех мыслимых IP-адресов. Выигрыш в памяти большой, а проигрыш в скорости невелик, если списки не слишком длинные, то есть совпадений псевдонимов немного.

Как же выбирать псевдонимы? Это и делает *хеш-функция*. В нашем примере хеш-функция h определена на множестве всех IP-адресов (из 2^{32} элементов), а значения её — целые числа от 1 до 250. Адрес x при этом хранится в ячейке с номером $n = h(x)$. Точнее, активные адреса x, для которых $h(x)$ одно и то же и равно n, связываются в список, и в ячейке массива с номером n хранится указатель на этот список. Если, как мы надеемся, большинство таких списков будут короткими, то поиск нужного адреса будет быстрым.

1.5.2. Семейства хеш-функций

Как найти хорошую хеш-функцию? Это должна быть действительно функция в том смысле, что данному адресу всегда ставится в соответствие один и тот же псевдоним. С другой стороны, эта функция должна быть достаточно «случайной», чтобы хорошо перемешивать разные адреса. Скажем, если псевдонимом IP-адреса считать его последний байт (при этом, например, $h(128.32.168.80) = 80$), то может возникнуть проблема, если у большинства клиентов в этом байте записано небольшое число (например, если компьютеры нумеруются в небольших локальных сетях по порядку). Если вместо последнего байта брать первый, тоже может получиться плохо (особенно если большинство клиентов из одного региона).

Обе этих функции были бы хороши, если бы адреса клиентов выбирались случайным образом из множества всех 2^{32} адресов (в том смысле, что средний размер списка в массиве был бы довольно мал). Проблема в том, что эти адреса нам неподконтрольны и могут быть распределены неравномерно.

Можно ли заранее подумать и раз навсегда выбрать хеш-функцию, которая будет хорошо работать на любых входных данных? Легко понять, что нет. Поскольку 2^{32} IP-адресам соответствуют всего 250 псевдонимов, то для любой хеш-функции найдутся как минимум $2^{32}/250 \approx 2^{24} \approx 16\,000\,000$ IP-адресов с одним и тем же хеш-значением; в терминологии хеш-функций это совпадение именуется *коллизией* (collision). И если большинство адресов будет из такого множества, то поиск будет медленным.

Выход из положения состоит в использовании случайности. Мы не выбираем хеш-функцию раз и навсегда, а при каждом запуске программы случайно выбираем её из некоторого класса функций. При этом, какие бы 250 адресов нам ни дали, для большинства функций этого класса коллизий будет мало. В результате при любом наборе клиентов у нас неплохие шансы на успех. (Злоумышленник, который как-то сможет узнать уже после начала работы программы, какую функцию мы выбрали, и соответственно будет подбирать адреса для атаки, всё же сможет поставить нас в тупик, но это уже гораздо более редкая ситуация.)

Как же выбрать подходящий класс функций? Нам поможет теория чисел. Идя ей навстречу, возьмём размер массива (n) не 250, а 257, поскольку 257 — *простое число*. (Казалось бы, при чём тут простые числа?) Будем считать IP-адрес четвёркой $x_1.x_2.x_3.x_4$ остатков по модулю 257. Чтобы определить хеш-функцию, фиксируем четвёрку остатков по модулю 257. Пусть это будут, скажем, 87, 23, 125 и 4. Тогда функция будет такой: $h(x_1.x_2.x_3.x_4) = 87x_1 + 23x_2 + 125x_3 + 4x_4 \bmod 257$.

Для произвольной четвёрки остатков $a = (a_1, a_2, a_3, a_4)$ получается функция

$$h_a(x_1.x_2.x_3.x_4) = \sum_{i=1}^{4} a_i \cdot x_i \bmod n.$$

Ниже мы покажем, что если выбирать четвёрку a случайно, то h_a с высокой вероятностью будет хорошей. Точнее, мы докажем такое свойство:

Свойство. *Пусть* $x_1.x_2.x_3.x_4$ *и* $y_1.y_2.y_3.y_4$ *— два различных IP-адреса. Если коэффициенты* $a = (a_1, a_2, a_3, a_4)$ *случайно выбираются из множества* $\{0, 1, \dots$ $\dots, n-1\}$ *(все* n *вариантов равновероятны, четыре выбора делаются независимо), то*

$$\mathrm{P}(h_a(x_1, x_2, x_3, x_4) = h_a(y_1, y_2, y_3, y_4)) = \frac{1}{n}.$$

Другими словами, вероятность коллизии $h_a(x) = h_a(y)$ такая же, как если бы значения для x и y выбирались случайно и независимо. Отсюда следует, что ожидаемое время поиска адреса в хеш-таблице будет небольшим. Более точно, фиксируем 250 адресов. Один из них обозначим через x. Длина списка, в который попадёт x, зависит от выбора хеш-функции. Именно эта длина определяет длительность поиска.

Каково среднее (по всем вариантам выбора хеш-функции) значение этой длины? Каждый из 249 остальных адресов с вероятностью $1/n = 1/257$ получает тот же самый псевдоним (хеш-значение), что и x. Значит, среднее количество адресов в x-списке будет $249/257$ плюс сам x, то есть меньше 2.[1]

Докажем теперь сформулированное свойство.

Доказательство. По условию $x_1.x_2.x_3.x_4 \neq y_1.y_2.y_3.y_4$. Пусть, скажем, $x_4 \neq$ $\neq y_4$. Вычислим $\mathrm{P}(h_a(x_1.x_2.x_3.x_4) = h_a(y_1.y_2.y_3.y_4))$, то есть вероятность того, что $\sum_{i=1}^{4} a_i \cdot x_i \equiv \sum_{i=1}^{4} a_i \cdot y_i \pmod{n}$. Перепишем последнее сравнение следующим образом:

$$\sum_{i=1}^{3} a_i \cdot (x_i - y_i) \equiv a_4 \cdot (y_4 - x_4) \pmod{n}. \qquad (1)$$

Функция h_a определяется случайным выбором четвёрки $a = (a_1, a_2, a_3, a_4)$. Допустим, мы уже выбрали случайно a_1, a_2 и a_3 и теперь хотим понять, для каких значений a_4 сравнение (1) выполняется. Поскольку a_1, a_2, a_3 уже зафиксированы, левая часть сравнения (1) равна некоторой константе c. Поскольку n простое и $x_4 \neq y_4$, у числа $x_4 - y_4$ есть обратное по модулю n. Сравнение (1) может выполняться только для $a_4 = c \cdot (y_4 - x_4)^{-1} \bmod n$, то есть ровно для одного значения a_4 из n возможных. □

Посмотрим ещё раз, что же мы получили. Поскольку мы не знаем, какие данные будут храниться в таблице, мы выбираем хеш-функцию случайно из некоторого семейства функций $\mathcal{H}$. В нашем примере

$$\mathcal{H} = \{h_a : a \in \{0, \dots, n-1\}^4\}.$$

Заметим, что две изначально предлагавшиеся хеш-функции, первый и последний байт адреса, тоже лежат в этом множестве, это $h_{(1,0,0,0)}$ и $h_{(0,0,0,1)}$.

---

[1]Формально говоря, мы пользуемся линейностью математического ожидания. Для каждого адреса, кроме x, у нас есть случайная величина, которая равна 1, если происходит x-y-коллизия, и равна нулю, если нет. Математическое ожидание этой величины равно вероятности коллизии, то есть $1/257$. Размер списка на единицу больше суммы этих величин, так что его математическое ожидание равно $1\frac{249}{257}$.

Чтобы выбрать из этого семейства функцию случайно и равновероятно, мы просто выбираем четыре случайных числа $a_1, \ldots, a_4$ по модулю n.

Без упоминания вероятностей доказанное свойство можно сформулировать так:

> Для любых двух различных x и y ровно $|\mathscr{H}|/n$ функций всего семейства присваивают элементам x и y одно и то же хеш-значение. (Здесь n — общее количество возможных хеш-значений.)

Семейства хеш-функций с таким свойством называют *универсальными*. С точки зрения коллизий двух элементов выбор хеш-функции из универсального семейства так же хорош, как и выбор случайной функции (вероятность коллизии для данной пары элементов $1/n$). Из этого следует оценка на среднее время поиска при использовании универсального семейства.

В нашем примере мы хранили IP-адреса, но, разумеется, можно хранить таким способом и что-то другое. Разумно выбрать в качестве размера таблицы простое число, которое чуть больше ожидающегося количества элементов в таблице (такое простое число всегда есть; на самом же деле, на практике часто выбирают размер таблицы с двукратным запасом). Если имеется не более $N = n^k$ возможных объектов хранения для некоторого k, то каждый объект можно рассматривать как последовательность из k остатков по модулю n. При этом семейство $\mathscr{H} = \{h_a : a \in \{0, \ldots, n-1\}^k\}$ будет универсальным семейством хеш-функций.

Упражнения

1.1. Покажите, что для любого основания $b \geqslant 2$ сумма любых трёх b-ичных цифр в b-ичной записи состоит не более чем из двух цифр.

1.2. Покажите, что для любого числа длина его двоичной записи не более чем в четыре раза превосходит длину его десятичной записи. Чему примерно равно отношение этих длин для очень больших чисел?

1.3. В d-ичном дереве у каждой его вершины не более d детей. Покажите, что глубина любого d-ичного дерева с n вершинами есть $\Omega(\log n / \log d)$. Можете ли вы привести точную формулу для минимальной глубины?

1.4. Докажите, что

$$\log(n!) = \Theta(n \log n).$$

(Подсказка: сравните $n!$ с n^n и $(n/2)^{n/2}$.)

1.5. В отличие от бесконечно убывающей геометрической прогрессии, *гармонический ряд* $1 + 1/2 + 1/3 + 1/4 + 1/5 + \ldots$ расходится:

$$\sum_{i=1}^{\infty} \frac{1}{i} = \infty.$$

Для больших n сумма первых n членов этого ряда примерно равна $\ln n + \gamma$, где $\ln$ — натуральный логарифм, а $\gamma = 0{,}57721\ldots$ — некоторая константа. Докажите более слабую оценку:

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n).$$

(Подсказка: зажмите каждый член суммы между двумя отрицательными степенями двойки.)

1.6. Докажите, что метод умножения чисел «в столбик» (с. 17) правильно работает для чисел в двоичной системе счисления.

1.7. За какое время рассмотренный нами рекурсивный алгоритм умножения чисел (с. 19) умножает число длины n на число длины m?

1.8. Докажите корректность алгоритма деления (с. 19) и докажите верхнюю оценку $O(n^2)$ на время его работы, где n — длина входных чисел.

1.9. Используя определение $x \equiv y \pmod N$ (как делимости $x - y$ на N), докажите правило подстановки

$$x \equiv x' \pmod N \quad \text{и} \quad y \equiv y' \pmod N \quad \Rightarrow \quad x + y \equiv x' + y' \pmod N,$$

а также соответствующее правило для умножения.

1.10. Докажите, что если $a \equiv b \pmod N$ и M делит N, то $a \equiv b \pmod M$.

1.11. Делится ли $4^{1356} - 9^{4824}$ на 35?

1.12. Чему равно $2^{2^{2011}} \bmod 3$?

1.13. Делится ли $5^{30000} - 6^{123456}$ на 31?

1.14. Допустим, нам нужно вычислить n-е число Фибоначчи F_n по модулю p. Видите ли вы быстрый способ сделать это? (Подсказка: вспомните упражнение 0.4.)

1.15. Приведите условия на x и c, необходимые и достаточные для возможности сокращения на x по модулю c: это значит, что для любых a и b из $ax \equiv bx \pmod c$ следует $a \equiv b \pmod c$.

1.16. Покажите, что вычисление $a^b \bmod c$ последовательным возведением в квадрат (и перемножением получившихся степеней двойки) требует такого же числа умножений, как алгоритм рис. 1.4. Приведите пример с $b > 10$, где это не оптимально и вычисление a^b может быть произведено за меньшее количество умножений.

1.17. Рассмотрим задачу вычисления x^y (нас интересует сама степень, а не её остаток по модулю). Мы знаем два алгоритма: итеративный алгоритм, делающий $y - 1$ умножений, и рекурсивный алгоритм, основанный на двоичном представлении y.

Сравните время работы данных двух алгоритмов, считая, что время умножения n-битового и m-битового чисел есть $O(mn)$.

1.18. Вычислите НОД(210, 588) двумя способами — с помощью алгоритма Евклида и разложив оба числа на множители.

1.19. Покажите, что $\text{НОД}(F_{n+1}, F_n) = 1$ при $n \geqslant 1$, где F_n — n-е число Фибоначчи.

1.20. Найдите обратные: 20 mod 79, 3 mod 62, 21 mod 91, 5 mod 23.

1.21. Сколько остатков по модулю $11^3 = 1331$ имеют обратные?

1.22. Докажите или опровергните такое утверждение: если у a есть обратное по модулю b, то и у b есть обратное по модулю a.

1.23. Покажите, что если у числа a есть обратное по модулю N, то обратное единственно (по модулю N).

1.24. Сколько чисел среди $\{0, 1, \ldots, p^n - 1\}$ имеют обратные по модулю p^n для простого p?

1.25. Вычислите $2^{125} \bmod 127$. (Подсказка: 127 — простое.)

1.26. Какова последняя цифра десятичной записи числа $17^{17^{17}}$? (Подсказка: для различных простых p и q, а также числа $a \not\equiv 0 \pmod{pq}$ в разделе 1.4.2 доказано равенство $a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$.)

1.27. Пусть в протоколе RSA (см. рис. 1.9) $p = 17$, $q = 23$, $N = 391$, $e = 3$. Какой нужно выбрать закрытый ключ d? Зашифруйте сообщение $x = 41$.

1.28. Пусть теперь $p = 7$ и $q = 11$ (рис. 1.9). Предложите подходящие друг к другу значения e и d.

1.29. Обозначим через $[m]$ множество $\{0, 1, \ldots, m - 1\}$. Для каждого из следующих семейств хеш-функций определите, является ли оно универсальным, а также сколько случайных битов понадобится, чтобы выбрать функцию из семейства.

(a) $\mathcal{H} = \{h_{a_1,a_2} : a_1, a_2 \in [m]\}$, где m — простое число и

$$h_{a_1,a_2} : [m]^2 \to [m], \quad h_{a_1,a_2}(x_1, x_2) = a_1 x_1 + a_2 x_2 \bmod m.$$

(b) $\mathcal{H}$ определяется так же, но теперь $m = 2^k$ — степень двойки.

(c) $\mathcal{H}$ — множество всех функций $f : [m] \to [m - 1]$.

1.30. Школьный алгоритм умножения двух n-битовых чисел x и y складывает n копий числа x с соответствующими сдвигами влево. Длина каждого числа с учётом сдвига не превосходит $2n$.

В этом упражнении мы складываем n чисел, содержащих m битов каждое, при помощи *схем* (circuits), вычисляющих результат *параллельно*. Интересовать нас будет главным образом глубина схемы, то есть длина самого длинного пути от входа схемы к её выходу. Время работы схемы (если каждый элемент работает фиксированное время) пропорционально её глубине.

При сложении двух m-битовых чисел «в столбик» для вычисления i-го бита ответа мы должны знать бит переноса из $(i - 1)$-го разряда. Глубина получающейся схемы для сложения, если строить её естественным способом, есть

$\Theta(m)$. Существуют, однако, схемы для сложения *с предвычислением переносов*, которые имеют глубину $O(\log m)$. (Посмотрите в википедии ‘carry lookahead adder’, если вас интересуют подробности.)

(a) Приняв на веру существование схем для сложения такой глубины, покажите, что можно вычислить сумму n чисел длины m схемой глубины $O((\log n)(\log m))$.

(b) Придумайте схему, которая берёт *три* m-битовых числа x, y, z и вычисляет два числа r и s, для которых $x + y + z = r + s$, причём каждый бит r и s вычисляется независимо от других. (Подсказка: одно из чисел будет представлять переносы.)

(c) Используя схемы из предыдущих пунктов, разработайте схему глубины $O(\log n)$ для умножения двух n-битовых чисел.

1.31. Допустим, мы хотим вычислить $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$.

(a) Сколько примерно битов (в терминах $\Theta(\cdot)$) в числе $N!$, если число N содержит n битов?

(b) Постройте алгоритм для вычисления $N!$ и оцените время его работы.

1.32. Число N называется *степенью* (power), если оно имеет вид q^k для положительных целых q и $k > 1$.

(a) Постройте эффективный алгоритм, определяющий, является ли заданное число N квадратом целого числа. Каково время работы вашего алгоритма?

(b) Покажите, что если $N = q^k$ (для положительных целых N, q, k), то либо $N = 1$, либо $k \leqslant \log N$.

(c) Постройте эффективный алгоритм, определяющий, является ли данное число N степенью, и оцените время его работы.

1.33. Постройте эффективный алгоритм вычисления *наименьшего общего кратного* (least common multiple) двух n-битовых чисел x и y, то есть минимального положительного целого числа, делящегося и на x, и на y. Каково время работы вашего алгоритма как функция от n?

1.34. На стр. 31 говорится, что, поскольку около $1/n$ всех n-битовых чисел являются простыми, в среднем достаточно проверить $O(n)$ случайных n-битовых чисел до первого простого. Вот как это можно доказать.

Допустим, что при бросании монеты орёл появляется с вероятностью p. Какое среднее количество подбрасываний понадобится, чтобы появился орёл (считая последнее, где выпал орёл)? (Подсказка: (вариант 1) докажите, что ответ равен $\sum\limits_{i=1}^{\infty} i(1-p)^{i-1}p$; (вариант 2) докажите, что $E = 1 + (1-p)E$, где E — искомое среднее.)

1.35. В этом упражнении мы докажем теорему Вильсона (Wilson): положительное целое число $N > 1$ является простым тогда и только тогда, когда

$$(N-1)! \equiv -1 \pmod{N}.$$

(a) Для простого p каждое число $x = 1, 2, \ldots, p-1$ имеет обратное по модулю p. Какие из них являются обратными к самим себе?

(b) Объединив числа и их обратные в пары, покажите, что $(p-1)!$ сравнимо с -1 по модулю p, если p простое.

(c) Покажите, что если N не является простым, то $(N-1)! \not\equiv -1 \pmod{N}$. (Подсказка: у N и $(N-1)!$ есть общий делитель.)

(d) В отличие от малой теоремы Ферма, теорема Вильсона даёт необходимое и достаточное условие простоты числа. Почему же мы не можем использовать её для проверки простоты?

1.36. *Квадратные корни.* В этом упражнении мы научимся извлекать квадратный корень по модулю простого числа p, для которого $p \equiv 3 \pmod{4}$, то есть $p = 4k+3$.

(a) Покажите, что для любого $x \not\equiv 0 \pmod{p}$ выполняется равенство $x^{4k+4} = x^2$.

(b) Число x называется *квадратным корнем* (square root) из a по модулю p, если $a \equiv x^2 \pmod{p}$. Покажите, что если у a есть квадратный корень по модулю $p = 4k+3$, то число a^{k+1} тоже будет квадратным корнем из a по модулю p.

1.37. *Китайская теорема об остатках.*

(a) Нарисуйте табличку с тремя колонками. В первую колонку запишите числа от 0 до 14. Во вторую запишите остатки чисел из первой колонки по модулю 3, в третью — по модулю 5. Что можно сказать о числах во второй и третьей колонках?

(b) Пусть p и q — различные простые числа. Докажите, что для любой пары остатков (j, k) по модулю p и по модулю q (соответственно) в диапазоне $0, \ldots, pq-1$ существует единственное число i, сравнимое с j по модулю p и сравнимое с k по модулю q. (Подсказка: покажите, что двум различным i не может соответствовать одна и та же пара (j, k), после чего посчитайте количество различных i и количество различных пар (j, k).)

(c) Зная i, легко найти пару (j, k) с указанными свойствами. В обратную сторону: найти i по (j, k) можно по следующей формуле:

$$i = \big(j \cdot q \cdot (q^{-1} \bmod p) + k \cdot p \cdot (p^{-1} \bmod q)\big) \bmod pq.$$

(d) Можете ли вы обобщить последние два пункта на случай трёх и более простых чисел?

1.38. Чтобы проверить, делится ли число, скажем, 562437487, на 3, можно просто сложить его цифры и проверить, делится ли на 3 полученное число ($5+6+2+4+3+7+4+8+7 = 46$ на 3 не делится).

Есть похожий критерий делимости на 11: разобьём цифры числа на пары, начиная справа (87, 74, 43, 62, 05) и проверим, делится ли на 11 сумма этих двузначных чисел. (Если полученное число слишком большое, можно повторить операцию.)

Как насчёт числа 37? В этом случае нужно проверять, делится ли на 37 сумма чисел, составленных из троек цифр исходного числа (487, 437, 562).

Аналогичный способ работает для любого простого p, отличного от 2 и 5: число n делится на p одновременно с суммой чисел, полученных разбиением n справа на группы фиксированной длины r (для каждого p своей).

(a) Найдите минимальное подходящее значение r для $p = 13$ и $p = 17$.

(b) Покажите, что минимальное подходящее r делит $p - 1$.

1.39. Приведите полиномиальный по времени алгоритм для вычисления $a^{b^c} \bmod p$ для целых положительных a, b, c и простого p.

1.40. Покажите, что если x является нетривиальным корнем из единицы по модулю N, то есть $x^2 \equiv 1 \pmod{N}$, но $x \not\equiv \pm 1 \pmod{N}$, то N является составным. (Например, $4^2 \equiv 1 \pmod{15}$, но $4 \not\equiv \pm 1 \pmod{15}$, и поэтому 4 является нетривиальным корнем из 1 по модулю 15. Следовательно, число 15 составное.)

1.41. *Квадратичные вычеты.* Число a называется *квадратичным вычетом* (quadratic residue) по модулю N, если существует такое x, что $a \equiv x^2 \pmod{N}$.

(a) Пусть N — нечётное простое число, а a — ненулевой квадратичный вычет по модулю N. Покажите, что в множестве $\{0, 1, \ldots, N-1\}$ уравнение $x^2 \equiv a \pmod{N}$ имеет ровно два решения.

(b) Покажите, что если N — нечётное простое число, то среди N остатков по модулю N имеется ровно $(N+1)/2$ квадратичных вычетов по модулю N.

(c) Покажите, что простота N существенна: приведите пример положительных целых чисел a и N, для которых сравнение $x^2 \equiv a \pmod{N}$ имеет более двух решений в множестве $\{0, 1, \ldots, N-1\}$.

1.42. Предположим, что в криптосистеме RSA (рис. 1.9) вместо составного $N = pq$ использовалось бы простое число p. Сообщение $m \bmod p$ кодировалось бы как $m^e \bmod p$. Покажите, что такая криптосистема ненадёжна, построив эффективный алгоритм дешифрования, то есть алгоритм, который по p, e и $m^e \bmod p$ вычисляет $m \bmod p$.

1.43. В криптосистеме RSA открытый ключ Боба (N, e) доступен всем. Допустим, что $e = 3$ и Ева каким-то образом узнала закрытый ключ d. Покажите, что после этого Ева легко может разложить N на множители.

1.44. Алиса и три её друга используют криптосистему RSA. При этом её друзья используют открытые ключи $(N_i, 3)$ с возведением в степень 3, и $N_i = p_i q_i$ для случайно выбранных n-битовых простых чисел p_i и q_i. Покажите, что если Алиса пошлёт одно и то же n-битовое сообщение M всем троим, то перехвативший все три закодированных сообщения (и знающий открытые ключи) сможет быстро восстановить M. (Подсказка: можно воспользоваться упражнением 1.37.)

1.45. *RSA и цифровая подпись.* В криптосистеме RSA у каждого участника есть открытый ключ $P = (N, e)$ и закрытый ключ d. *Схема цифровой подписи*

(digital signature scheme) основана на двух алгоритмах — Sign и Verify. Алгоритм Sign получает сообщение и закрытый ключ и выдаёт подпись σ. Алгоритм Verify получает открытый ключ (N, e), подпись σ и сообщение M; он проверяет, могла ли подпись σ быть получена посредством алгоритма Sign (вызванного для сообщения M и закрытого ключа, соответствующего открытому ключу (N, e)).

(a) Для чего нужны цифровые подписи?

(b) С помощью протокола RSA цифровая подпись может быть реализована следующим образом: $\text{Sign}(M, d) = M^d \bmod N$. Покажите, что любой знающий открытый ключ (N, e) может выполнить проверку $\text{Verify}((N, e), M^d, M)$, то есть проверить, что подпись была создана алгоритмом Sign. Постройте такой алгоритм и докажите его корректность.

(c) Выберите небольшие числа для протокола RSA: модуль $N = pq$, открытый ключ e и закрытый ключ d. Желательно, чтобы в десятичной записи N было не более четырёх цифр, чтобы не пользоваться калькулятором. Теперь создайте цифровую подпись RSA для вашего имени. Для этого понадобится разбить имя на блоки и фиксировать соответствие между блоками и числами по модулю N. Сделайте это, после чего вычислите, какие числа $m_1, m_2, \ldots, m_k$ соответствуют вашему имени. Подпишите первое число, вычислив $m_1^d \bmod N$. И наконец, проверьте, что $(m_1^d)^e \equiv m_1 \pmod{N}$.

(d) Алиса хочет написать сообщение от имени Боба. Она знает, что открытый ключ Боба — это пара (391, 17). В какую степень ей нужно возвести её сообщение?

1.46. *Цифровая подпись, продолжение.*

(a) Цифровая подпись предыдущего упражнения использует те же действия, что и декодирование, что может привести к неприятностям. Покажите, что если Боб готов подписывать что угодно, то Ева может воспользоваться этим и расшифровать любое сообщение Алисы Бобу.

(b) Пусть Боб теперь более осторожен и отказывается подписывать сообщения, если вычисленная им подпись подозрительно похожа на текст (считаем, что случайно выбранное число от 1 до $N - 1$ похожа на текст с очень малой вероятностью). Покажите, что Ева тем не менее сможет расшифровать любое сообщение для Боба.

Глава 2
Метод «разделяй и властвуй»

Метод *«разделяй и властвуй»* (divide-and-conquer) состоит в следующем:

1. Задача разбивается на несколько более простых *подзадач* (subproblems) того же типа.

2. Подзадачи решаются рекурсивно.

3. Из ответов для подзадач строится ответ для исходной задачи.

Эффективность такого подхода определяется тем, насколько быстро мы можем (1) делить задачу на подзадачи; (2) решать подзадачи, не поддающиеся дальнейшему делению и (3) собирать решение задачи из решений подзадач.

Для начала рассмотрим рекурсивный алгоритм умножения многозначных чисел, который гораздо быстрее «школьного».

2.1. Умножение чисел

Выдающийся немецкий математик Карл Фридрих Гаусс (1777—1855) заметил, что хотя формула для произведения двух комплексных чисел

$$(a+bi)(c+di) = ac - bd + (bc+ad)i$$

содержит *четыре* умножения вещественных чисел, можно обойтись и *тремя*: вычислим ac, bd и $(a+b)(c+d)$ и воспользуемся тем, что

$$bc + ad = (a+b)(c+d) - ac - bd.$$

Кажется, что выигрыш невелик: ускорение поглощается константой в O-обозначениях. Однако такое скромное улучшение становится очень важным при *рекурсивном использовании*.

Вспомним, что нас интересует умножение многозначных чисел. Рассмотрим два n-битовых числа x и y и допустим, что n есть степень двойки (если это не так, допишем слева недостающие нули; n при этом увеличится не более чем вдвое). Разобьём каждое из чисел x и y на две половины длины $n/2$:

$$x = \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R,$$
$$y = \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R.$$

Например, если $x = 10110110_2$ (двойка в нижнем индексе означает двоичную систему счисления), то $x_L = 1011_2$, $x_R = 0110_2$ и $x = 1011_2 \cdot 2^4 + 0110_2$.

Теперь заметим, что

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Будем вычислять xy согласно правой части равенства. Сложение и домножение на степень двойки (сдвиг влево) производятся за линейное время от длины чисел. Произведения $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$ можно вычислить за четыре рекурсивных вызова. Таким образом, наш алгоритм, умножая два числа длины n, сперва рекурсивно перемножает четыре пары $n/2$-битовых чисел (четыре подзадачи вдвое меньшего размера), после чего собирает из ответов произведение xy за время $O(n)$. Обозначив через $T(n)$ общее время работы алгоритма на n-битовых числах, приходим к следующему *рекуррентному соотношению* (recurrence relation):

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n).$$

Нетрудно показать, что отсюда следует $T(n) = O(n^2)$ (см. следующий раздел 2.2). Таким образом, мы ничего не выиграли по сравнению с умножением в столбик.

Для ускорения алгоритма нам и понадобится замечание Гаусса (точнее, его аналог для многозначных чисел). Вычисляя xy, можно обойтись тремя произведениями $(n/2)$-битовых чисел: $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$. В самом деле,

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

Время работы соответствующего алгоритма (рис. 2.1), удовлетворяет соотношению[1]

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n).$$

Вместо четырёх умножений мы (на каждом уровне рекурсии!) обошлись тремя. Как мы сейчас увидим, это сокращает время работы алгоритма до $O(n^{1{,}59})$.

Чтобы оценить время работы алгоритма, посмотрим на дерево рекурсии (рис. 2.2). На каждом следующем уровне подзадачи имеют вдвое меньший размер. На уровне $\log_2 n$ размер подзадач доходит до 1 и рекурсия заканчивается. Высота дерева, таким образом, равна $\log_2 n$. Каждая задача разбивается на три подзадачи вдвое меньшего размера, значит, на k-м уровне дерева находится 3^k задач размера $n/2^k$. Каждая подзадача требует линейного времени для дальнейшего разбиения и для сборки ответа из ответов к подзадачам. Таким образом, общее количество операций на k-м уровне есть

$$3^k \cdot O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \cdot O(n).$$

[1] Строго говоря, рекуррентное соотношение имеет вид

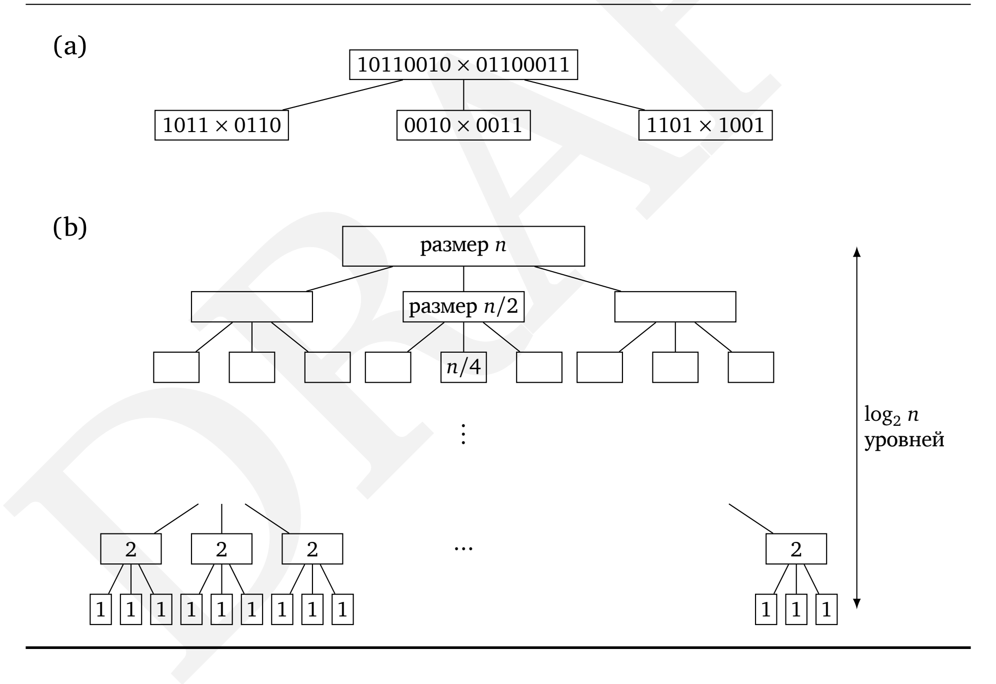
$$T(n) = 3T\left(\frac{n}{2} + 1\right) + O(n),$$

поскольку числа $(x_L + x_R)$ и $(y_L + y_R)$ могут иметь длину $n/2 + 1$. Оно даёт ту же асимптотическую оценку, что и наш упрощённый вариант.

Рис. 2.1. Умножение целых чисел методом «разделяй и властвуй».

функция $\textsc{Multiply}(x, y)$
{Вход: положительные целые числа x и y.}
{Выход: xy.}
$n \leftarrow \max($размер x, размер $y)$
если $n = 1$: вернуть xy
$x_L, x_R \leftarrow$ левые $\lceil n/2 \rceil$, правые $\lfloor n/2 \rfloor$ битов x
$y_L, y_R \leftarrow$ левые $\lceil n/2 \rceil$, правые $\lfloor n/2 \rfloor$ битов y
$P_1 \leftarrow \textsc{Multiply}(x_L, y_L)$
$P_2 \leftarrow \textsc{Multiply}(x_R, y_R)$
$P_3 \leftarrow \textsc{Multiply}(x_L + x_R, y_L + y_R)$
вернуть $P_1 \times 2^{2\lfloor n/2 \rfloor} + (P_3 - P_1 - P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$

Рис. 2.2. Умножение чисел методом «разделяй и властвуй»: (a) задача, разбитая на три подзадачи; (b) дерево рекурсии.



На самом верхнем уровне ($k = 0$) получаем $O(n)$, на нижнем ($k = \log_2 n$) получаем $O(3^{\log_2 n})$ или $O(n^{\log_2 3})$. При движении сверху вниз количество всех операций данного уровня растёт как *геометрическая прогрессия* со знаменателем $3/2$, от $O(n)$ до $O(n^{\log_2 3})$. Сумма такой прогрессии с точностью до постоянного множителя оценивается последним членом прогрессии (упражнение 0.2). Значит, общее время работы есть $O(n^{\log_2 3}) \leqslant O(n^{1{,}59})$.

Если бы не трюк Гаусса, то коэффициент ветвления дерева (количество детей у внутренней вершины) был бы равен 4. У дерева тогда было бы $4^{\log_2 n} = n^2$ листьев, и время работы алгоритма было бы квадратичным. Так что коэффициент ветвления в методе «разделяй и властвуй» очень важен.

На практике нет смысла спускаться в дереве рекурсии до однобитовых чисел. В большинстве процессоров умножение 16- или 32-битовых чисел является элементарной операцией, поэтому на этом уровне можно (и нужно) воспользоваться встроенной командой умножения.

А существует ли ещё более быстрый алгоритм? Оказывается, что да; он тоже основан на методе «разделяй и властвуй» (см. раздел 2.6 о быстром преобразовании Фурье).

2.2. Рекуррентные соотношения

Представим себе алгоритм, работающий по методу «разделяй и властвуй». Пусть он сводит данную задачу размера n к a подзадачам размера n/b и из найденных ответов строит ответ для исходной задачи. Будем считать, что на разбиение и сборку уходит время $O(n^d)$ (в рассмотренном выше алгоритме умножения $a=3$, $b=2$, $d=1$). Рекуррентное соотношение на время работы такого алгоритма:

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d).$$

Следующий результат даёт явную оценку для таких функций.

Основная теорема. *Пусть* $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ *для некоторых констант* $a > 0$, $b > 1$, $d \geqslant 0$. *Тогда*

$$T(n) = \begin{cases} O(n^d), & \text{если } d > \log_b a, \\ O(n^d \log n), & \text{если } d = \log_b a, \\ O(n^{\log_b a}), & \text{если } d < \log_b a. \end{cases}$$

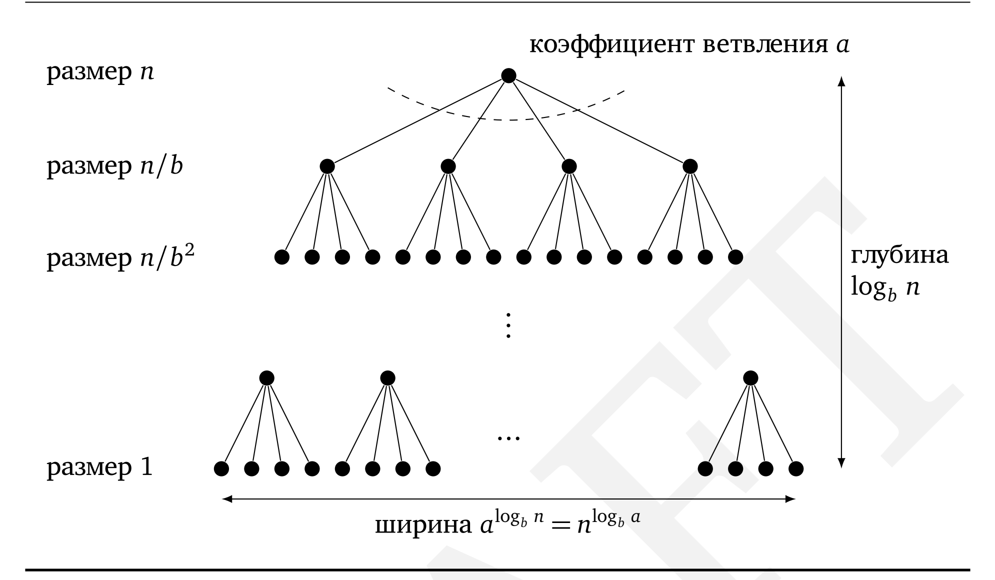
Доказательство. Для простоты будем считать, что n есть степень b. Это не сказывается на асимптотике, так как n отличается от ближайшей сверху степени b не более чем в b раз (упражнение 2.2); ясно, что при уменьшении n высота дерева и его размер могут только уменьшиться.

Дерево работы алгоритма имеет $\log_b n$ уровней, поскольку с увеличением номера уровня размер подзадач уменьшается в b раз. Коэффициент ветвления дерева есть a, так что k-й уровень содержит a^k подзадач размера n/b^k (рис. 2.3). Количество операций, которое алгоритм производит на этом уровне, есть

$$t_k = a^k \cdot O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d) \cdot \left(\frac{a}{b^d}\right)^k.$$

Последовательность $\{t_k\}$ (где k меняется от 0 до $\log_b n$), таким образом, является геометрической прогрессией со знаменателем a/b^d. Её первый член t_0

Рис. 2.3. Задача размера n разбивается на a подзадач размера n/b.



равен $O(n^d)$; последний член, при $k = \log_b n$, соответствует $a^{\log_b n} = n^{\log_b a}$ задачам размера 1, то есть равен $O(n^{\log_b a})$.

Оценка на сумму членов геометрической прогрессии зависит от её знаменателя (упражнение 0.2). Получаем три варианта, соответствующие трём случаям в теореме:

1. *Знаменатель меньше* 1.
Прогрессия убывает, и её сумма оценивается через первый член: $O(n^d)$.

2. *Знаменатель больше* 1.
Прогрессия возрастает; сумма оценивается через последний член $O(n^{\log_b a})$.

3. *Знаменатель равен* 1.
Тогда все $O(\log n)$ членов прогрессии равны, и её сумма есть $O(n^d \log n)$. □

Двоичный поиск

Стандартный алгоритм, основанный на методе «разделяй и властвуй»,— алгоритм двоичного поиска: для нахождения элемента k в отсортированном массиве элементов $z[0, 1, \ldots, n-1]$ мы сперва сравниваем k с $z[n/2]$ и в зависимости от результата рекурсивно вызываем алгоритм либо для первой части $z[0, 1, \ldots, n/2-1]$, либо для второй части $z[n/2, \ldots, n-1]$. Рекуррентное соотношение в данном случае имеет вид $T(n) = T(\lceil n/2 \rceil) + O(1)$. Воспользовавшись доказанной выше теоремой при $a = 1$, $b = 2$, $d = 0$, получаем, что $T(n) = O(\log n)$.

2.3. Сортировка слиянием

Метод «разделяй и властвуй» можно применить для сортировки массива: разделим массив на две части, рекурсивно отсортируем каждую из них, после чего *сольём* (merge) отсортированные части.

функция MergeSort($a[1\ldots n]$)
{Вход: массив чисел $a[1\ldots n]$.}
{Выход: отсортированный массив.}
если $n > 1$:
  вернуть Merge(MergeSort($a[1\ldots\lceil n/2\rceil]$), MergeSort($a[\lceil n/2\rceil+1\ldots n]$))
иначе:
  вернуть a

Корректность этого алгоритма очевидна (если, конечно, мы правильно запрограммировали слияние). Как же слить два отсортированных массива $x[1\ldots k]$ и $y[1\ldots l]$ в один массив $z[1\ldots k+l]$? Ясно, что первым элементом массива z будет меньшее из чисел $x[1]$ и $y[1]$. Оставшаяся часть z может быть заполнена рекурсивно.

функция Merge($x[1\ldots k]$, $y[1\ldots l]$)
если $k = 0$: вернуть $y[1\ldots l]$
если $l = 0$: вернуть $x[1\ldots k]$
если $x[1] \leqslant y[1]$:
  вернуть $x[1] \circ$ Merge($x[2\ldots k]$, $y[1\ldots l]$)
иначе:
  вернуть $y[1] \circ$ Merge($x[1\ldots k]$, $y[2\ldots l]$)

Через $\circ$ мы обозначаем конкатенацию массивов (приписывание их друг к другу). Алгоритм Merge тратит время $O(1)$ на каждый рекурсивный вызов (если он реализован аккуратно и не копирует элементы массивов без надобности). Общее время его работы $O(k+l)$, то есть линейное. Отсюда получаем рекуррентное соотношение для времени сортировки слиянием n элементов:

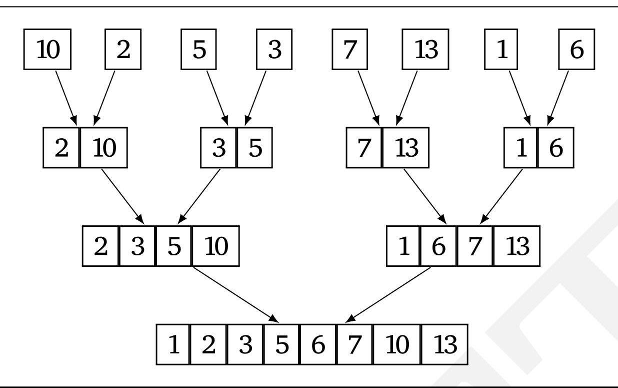
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

а значит, $T(n) = O(n \log n)$.

Куда уходит это время? В основном на слияние массивов, которое начинается с массивов единичного размера. Они сливаются в массивы размера 2, которые потом сливаются в массивы размера 4 и так далее. Пример работы алгоритма приведён на рис. 2.4.

Имея это в виду, напишем нерекурсивную версию алгоритма MergeSort. В каждый момент будем поддерживать множество «активных» массивов (упорядоченных кусков исходного; изначально куски состоят из одного элемента), которые попарно сливаются и образуют следующее множество активных массивов. Массивы можно хранить в очереди, из начала которой мы будем извлекать два массива, сливать их и помещать полученный массив в конец очереди. Вот что при этом получается (операция Inject добавляет элемент в конец очереди, а операция Eject извлекает элемент из

Рис. 2.4. Вызовы процедуры Merge в алгоритме MergeSort.

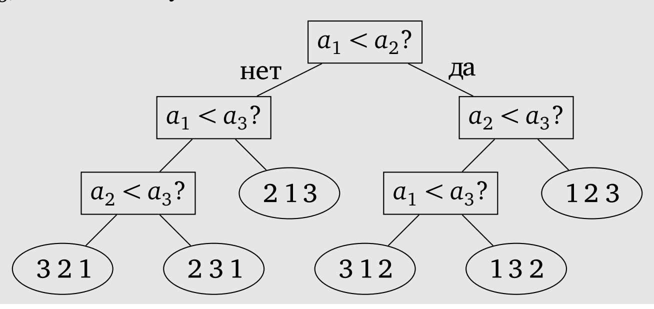


начала очереди):

```
функция IterativeMergeSort(a[1...n])
{Вход: массив чисел a[1...n].}
{Выход: отсортированный массив.}
Q ← [ ] {пустая очередь}
для i от 1 до n:
  Inject(Q, [a_i])
пока |Q| > 1:
  Inject(Q, Merge(Eject(Q), Eject(Q)))
вернуть Eject(Q)
```

Нижняя оценка $n \log n$ для сортировки

Работу алгоритма сортировки можно изобразить в виде дерева. Например, алгоритм на рисунке сортирует по возрастанию массив из трёх элементов a_1, a_2, a_3. Первым делом он сравнивает a_1 с a_2. Если a_1 не меньше a_2, он сравнивает его с a_3, в противном же случае сравнивает a_2 с a_3 и так далее. В конце концов мы приходим в лист, где уже известен правильный порядок трёх элементов (как перестановка чисел 1, 2, 3). Например, если $a_2 < a_1 < a_3$, то в листе будет записано «2 1 3».



(Алгоритм сортировки слиянием тоже можно представить в таком виде, если обращать внимание только на сравнения, а не на перемещение элементов; в момент окончания алгоритма порядок элементов однозначно определяется результатами выполненных сравнений.)

Глубина (depth) данного дерева, то есть количество сравнений на самом длинном пути от корня к листу (в примере глубина равна 3), является нижней оценкой времени работы алгоритма в худшем случае.

Рассматривая алгоритмы сортировки такого типа, можно показать, что *алгоритм сортировки слиянием оптимален*: для сортировки n элементов в худшем случае требуется $\Omega(n \log n)$ сравнений.

В самом деле, рассмотрим дерево сортировки n элементов. Каждый его лист помечен перестановкой множества $\{1, 2, \ldots, n\}$. Более того, нетрудно видеть, что для *каждой* перестановки должен найтись такой лист (поскольку соответствующая перестановка может быть подана на вход). Значит, у рассматриваемого нами дерева есть хотя бы $n!$ листьев.

У двоичного дерева глубины d может быть не более 2^d листьев. Таким образом, глубина дерева (и сложность нашего алгоритма) не меньше $\log(n!)$.

Известно, что $\log(n!) \geqslant c \cdot n \log n$ для некоторой константы $c > 0$. Есть несколько способов это доказать. Самый простой — заметить, что $n! \geqslant (n/2)^{(n/2)}$, поскольку в произведении $1 \cdot 2 \cdot \ldots \cdot n$ есть хотя бы $n/2$ множителей, не меньших $n/2$, и взять логарифм обеих частей неравенства. Другой способ даёт формула Стирлинга: $n! \approx \sqrt{\pi\left(2n + \frac{1}{3}\right)} \cdot n^n \cdot e^{-n}$. Так или иначе, мы установили, что дерево сравнений, сортирующее n элементов, требует $\Omega(n \log n)$ сравнений в худшем случае, а значит, алгоритм сортировки слиянием оптимален.

Тонкость в данном рассуждении заключается в том, что оно применимо только к *алгоритмам сортировки сравнениями* (такой алгоритм может сравнивать числа и их копировать, но никак не использует их внутреннюю структуру). Может быть, существуют алгоритмы, более хитро обращающиеся с числами и имеющие линейное время работы? Да, существуют: стандартный пример — сортировка чисел из небольшого по размеру диапазона (упражнение 2.20).

2.4. Медианы

Медианой (median) массива чисел называют число, которое будет стоять посередине, если массив отсортировать по возрастанию (неубыванию). Например, медианой массива [45, 1, 10, 30, 25] является число 25. Если массив имеет чётную длину, то есть две позиции, которые можно назвать средними; договоримся в такой ситуации выбирать меньшую из них (левую).

Медиана, как и среднее арифметическое, даёт некоторое представление о всём массиве чисел, но это разные вещи. Например, и медиана, и среднее значение массива из ста единиц равны 1. Однако если заменить одну из единиц на число 10000, то среднее станет больше 100, а медиана не изменится.

Ещё можно отметить, что медиана всегда будет элементом массива, в отличие от среднего.

Найти медиану n чисел можно с помощью сортировки. Однако это требует времени $\Omega(n \log n)$. Нет ли линейного алгоритма? Ведь сортировка делает много лишнего: нам нужен только средний элемент.

Оказывается, что такой алгоритм есть, и мы его построим. Для начала отметим, что при построении рекурсивного алгоритма часто помогает переход к *более общей* (и тем самым более трудной!) задаче. Дело в том, что обобщение позволяет использовать более сильное предположение индукции.

В нашем случае вместо нахождения медианы мы рассмотрим задачу *нахождения k-го по величине элемента*. В ней дан массив чисел S и число k; надо найти элемент, который будет стоять на k-м месте, если отсортировать элементы S по возрастанию. Например, при $k = 1$ ищется минимум, а при $k = \lfloor |S|/2 \rfloor$ — медиана.

2.4.1. Вероятностный алгоритм

Мы рассмотрим вероятностный алгоритм поиска k-го по величине элемента, основанный на методе «разделяй и властвуй».

Пусть v — элемент массива S. Разобьём S на три части: элементы, меньшие v, равные v (их может быть несколько) и бóльшие v. Назовём их S_L, S_v и S_R. Например, при разбиении массива

S:

2	36	5	21	8	13	11	20	5	4	1

по элементу $v = 5$ получаются

S_L:

2	4	1

S_v:

5	5

S_R:

36	21	8	13	11	20

Поиск нужного элемента в исходном массиве легко может быть сведён к поиску в одной из трёх частей. Например, *восьмой* по величине элемент S — это *третий* по величине элемент S_R, поскольку $|S_L| + |S_v| = 5$. В общем случае достаточно сравнить k с размерами частей, чтобы понять, в какой из частей находится нужный нам элемент:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k), & \text{если } k \leqslant |S_L|, \\ v, & \text{если } |S_L| < k \leqslant |S_L| + |S_v|, \\ \text{selection}(S_R, k - |S_L| - |S_v|), & \text{если } k > |S_L| + |S_v|. \end{cases}$$

Части S_L, S_v, S_R можно сформировать за линейное время, и это можно сделать на том же месте, где был массив S (упражнение 2.15). После этого можно рекурсивно вызвать алгоритм для одной из частей S_L или S_R: мы переходим от массива размера $|S|$ к массиву размера $\max\{|S_L|, |S_R|\}$.

Как выбирать v? Это надо делать быстро; в то же время хотелось бы, чтобы размер массива быстро уменьшался. В идеале массив уменьшается вдвое: $|S_L|, |S_R| \approx |S|/2$. Если бы мы могли гарантировать такое уменьшение, то для времени работы получили бы соотношение

$$T(n) = T\left(\frac{n}{2}\right) + O(n),$$

то есть оно было бы линейно. Но для этого в качестве v нужно уметь выбирать медиану, а мы пока только учимся это делать. Поступим же мы гораздо проще: *будем выбирать v случайно*.[1]

2.4.2. Анализ времени работы

Время работы алгоритма, конечно же, зависит от того, как выбирается v. Если нам фатально не везёт и каждый раз мы выбираем в качестве v максимальный (или минимальный) элемент, то размер массива за один шаг будет уменьшаться всего на единицу. (В примере выше мы бы выбрали сначала $v = 36$, потом $v = 21$ и так далее.) В таком худшем (хотя и маловероятном) случае алгоритму потребуется около

$$n + (n-1) + \ldots + \frac{n}{2} = \Theta(n^2)$$

операций для вычисления медианы. Так же маловероятен и *самый хороший случай*, когда v каждый раз делит массив ровно пополам; при этом время работы будет $O(n)$. К какой же из этих крайностей ближе *среднее время* работы? К счастью, оно довольно близко к оценке в лучшем случае.

Чтобы формально различать «удачный» и «неудачный» выбор v, будем называть элемент v хорошим, если при сортировке массива S он попадёт на место с номером от $|S|/4$ до $3|S|/4$. При разбиении массива по хорошему элементу размеры массивов S_L и S_R будут не больше $3|S|/4$. Хороших элементов довольно много — как раз половина, то есть случайно выбранный элемент v окажется хорошим с вероятностью $1/2$. Каково среднее количество шагов до появления хорошего элемента? Вот ответ на этот вопрос, переформулированный в более привычных терминах (см. также упражнение 1.34):

Лемма. *Математическое ожидание количества подбрасываний монетки до первой решки* (*включительно*) *равно* 2.

Доказательство. Обозначим эту величину E. Если решка выпала сразу, то дальше бросать не надо, иначе (с вероятностью $1/2$) мы возвращаемся в исходную ситуацию. Значит, $E = \frac{1}{2} + \frac{1+E}{2}$, откуда $E = 2$. □

Совсем грубо можно сказать, что в среднем после двух разбиений массива его размер уменьшится хотя бы на четверть, что даёт такое рекуррентное соотношение:

$$T(n) \leqslant T\left(\frac{3n}{4}\right) + O(n).$$

Аккуратное рассуждение требует многих уточнений: поскольку время работы зависит от входа (числа одинаковых элементов), через $T(n)$ надо обозначать максимум (по всех входам длины не больше n) среднего (по случайным выборам) времени работы. Рекуррентная оценка получается взятием среднего в обеих частях следующего неравенства (мы также используем

[1] Существует и детерминированный алгоритм поиска медианы за линейное время, но он сложнее, и мы его здесь не рассматриваем.

Алгоритм быстрой сортировки

Рассмотренные нами алгоритмы сортировки и нахождения медианы оба основаны на методе «разделяй и властвуй», но действуют по-разному. Алгоритм сортировки слиянием разбивает массив на две части, совершенно не заботясь о том, что попадёт в эти части. Основная работа производится при слиянии рекурсивно отсортированных массивов. Алгоритм нахождения медианы разбивает массив более тщательно, что позволяет ему ничего не делать после рекурсивного вызова.

Алгоритм быстрой сортировки (quicksort) разбивает массив точно так же, как алгоритм нахождения медианы. После сортировки полученных подмассивов двумя рекурсивными вызовами весь исходный массив оказывается отсортированным. Время работы данного алгоритма в худшем случае, как и для алгоритма нахождения медианы, есть $\Theta(n^2)$. Можно доказать (упражнение 2.24), что его *среднее* время работы $O(n \log n)$. Более того, на практике алгоритм быстрой сортировки часто оказывается эффективнее других при сортировке массивов большого размера.

линейность среднего: *среднее суммы равно сумме средних её слагаемых*):

$$\text{время работы на входе размера до } n \leqslant \\ \leqslant \left(\text{время работы на входе размера до } \frac{3n}{4}\right) + \\ + \left(\text{время на уменьшение размера массива до } \frac{3n}{4}\right).$$

Среднее второго слагаемого оценивается исходя из того, что на каждом уровне рекурсии требуется $O(n)$ операций, а среднее число уровней есть $O(1)$ (строго говоря, вероятность успеха не равна $1/2$, а может быть и больше, но можно доказать, что это только уменьшает среднее в лемме выше). Из данного неравенства следует, что $T(n) = O(n)$: для *любого* входа среднее время работы алгоритма на этом входе линейно.

2.5. Умножение матриц

Произведением двух $(n \times n)$-матриц X и Y называется $(n \times n)$-матрица $Z = XY$, где

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

Другими словами, Z_{ij} есть скалярное произведение i-й строки матрицы X на j-й столбец матрицы Y:

i X × j Y = (i, j) Z

Умножение матриц не коммутативно (XY и YX могут различаться), но ассоциативно: $(XY)Z = X(YZ)$.

«Наивный» алгоритм, основанный на этой формуле, вычисляет произведение за $O(n^3)$ операций: на каждое из $O(n^2)$ значений требуется $O(n)$ операций. Довольно долго наивный алгоритм казался асимптотически оптимальным; было даже доказано, что в некоторых моделях вычислений не существует более быстрых алгоритмов. Поэтому более эффективный алгоритм, предложенный немецким математиком Фолькером Штрассеном в 1969 году, оказался неожиданным. Он использует метод «разделяй и властвуй».[1]

Задачу умножения матриц достаточно легко разбить на подзадачи, поскольку произведение можно составлять из *блоков*. Разобьём каждую из матриц X и Y на четыре блока размера $\frac{n}{2} \times \frac{n}{2}$:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Тогда их произведение выражается в терминах этих блоков по обычной формуле умножения матриц (упражнение 2.11):

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Вычислив рекурсивно восемь произведений AE, BG, AF, BH, CE, DG, CF, DH и просуммировав их за время $O(n^2)$, мы вычислим необходимое нам произведение матриц. Соответствующее рекуррентное соотношение на время работы алгоритма,

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2),$$

даёт $O(n^3)$, так что такой рекурсивный алгоритм ничуть не лучше наивного. Однако его можно ускорить с помощью алгебраического трюка: для вычисления произведения XY достаточно перемножить *семь* пар матриц размера $\frac{n}{2} \times \frac{n}{2}$, после чего хитрым образом (и как только Штрассен догадался?) получить ответ:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix},$$

где

$$\begin{aligned} P_1 &= A(F - H), & P_5 &= (A + D)(E + H), \\ P_2 &= (A + B)H, & P_6 &= (B - D)(G + H), \\ P_3 &= (C + D)E, & P_7 &= (A - C)(E + F), \\ P_4 &= D(G - E). \end{aligned}$$

Теперь для времени работы получаем соотношение $T(n) = 7T(n/2) + O(n^2)$, так что $T(n) = O(n^{\log_2 7}) \approx O(n^{2{,}81})$.

[1]На практике данный алгоритм даёт выигрыш только при достаточно бóльших n. Асимптотически и он не оптимален; лучшую оценку $O(n^{2{,}37\dots})$ даёт алгоритм Копперсмита—Винограда и его модификации. Более того, некоторые предполагают, что произведение матриц может быть вычислено за время $O(n^{2+o(1)})$. — *Прим. перев.*

2.6. Быстрое преобразование Фурье

Используя метод «разделяй и властвуй», мы научились быстро умножать числа и матрицы. Научимся теперь быстро умножать *многочлены* (полиномы). Произведение двух многочленов степени d есть многочлен степени $2d$, например,

$$(1+2x+3x^2)\cdot(2+x+4x^2)=2+5x+12x^2+11x^3+12x^4.$$

Формально, произведение двух многочленов $A(x)=a_0+a_1x+\ldots+a_dx^d$ и $B(x)=b_0+b_1x+\ldots+b_dx^d$ есть многочлен $C(x)=A(x)\cdot B(x)=c_0+c_1x+\ldots$ $\ldots+c_{2d}x^{2d}$, где

$$c_k=\sum_{i=0}^{k} a_i b_{k-i}$$

(при $i>d$ полагаем a_i и b_i равными нулю). Значение c_k можно вычислить за $O(k)$ арифметических операций, а все $2d+1$ коэффициентов можно найти за $O(d^2)$ операций. *Но нельзя ли перемножить многочлены быстрее*?

Оказывается, что можно. Алгоритм, который при этом используется, — *быстрое преобразование Фурье* — произвёл революцию в цифровой обработке сигналов (да и вообще сделал её возможной). Этот важный для практического применения алгоритм (его текст приведён в разделе 2.6.4) мы рассмотрим подробно.

2.6.1. Альтернативное представление многочленов

Быстрый алгоритм умножения использует такое свойство многочленов:

Факт. *Многочлен степени d однозначно задаётся значениями в любых $d+1$ различных точках.*

В частности, как известно, любые две различные точки задают прямую (график многочлена первой степени).

Выберем произвольные $d+1$ точек $x_0, \ldots, x_d$. Многочлен степени не выше d имеет вид $A(x)=a_0+a_1x+\ldots+a_dx^d$, и его можно задать одним из двух способов:

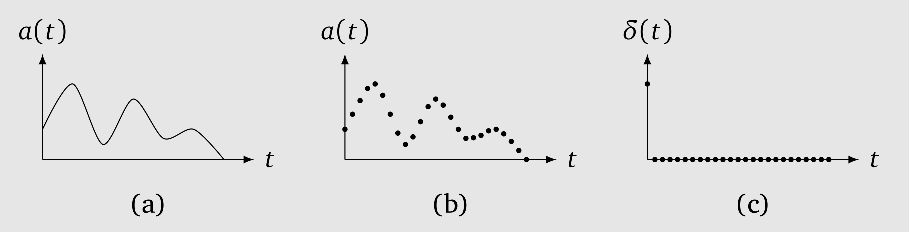
- коэффициентами $a_0, \ldots, a_d$,
- значениями $A(x_0), \ldots, A(x_d)$.

Второе представление более удобно для умножения: зная значения сомножителей в каждой из точек, легко вычислить значение произведения (число умножений равно числу точек, то есть пропорционально степени многочлена). Итак, *если многочлены задаются своими значениями, то их умножение занимает линейное время*.

Для чего нужно умножать многочлены?

Прежде всего, самые быстрые из известных алгоритмы умножения чисел основаны на умножении многочленов (задачи умножения чисел и многочленов родственны: достаточно заменить x на основание системы

счисления и следить за переносами). Но, пожалуй, наиболее важное применение умножения многочленов — *цифровая обработка сигналов* (signal processing). *Сигналом* (a) называется произвольная функция времени или координаты. Например, человеческий голос будет звуковым сигналом, отражающим колебания воздуха у рта говорящего. Картину ночного неба тоже можно считать сигналом (только двумерным), измеряя яркость как функцию координат.



(a) (b) (c)

Обрабатывая сигнал, мы дискретизируем его, беря значения через определённые промежутки времени (b). Посмотрим, как такой сигнал проходит через некоторую преобразующую его *систему*. Выходной сигнал такой системы называют *откликом*:

входной сигнал → СИСТЕМА → отклик

Важный класс составляют системы, которые обладают свойством *линейности* (отклик на сумму сигналов равен сумме откликов на каждый из них по отдельности) и *инвариантности по времени* (сдвинутый на время t сигнал приводит к сдвинутому на то же время отклику). Любая система, обладающая этими свойствами, полностью характеризуется реакцией на самое простое возможное воздействие — *единичный импульс* $\delta(t)$ (unit impulse), равный единице в момент времени $t = 0$ и нулю в остальное время (c). Чтобы убедиться в этом, рассмотрим задержанный импульс $\delta(t-i)$, равный единице в момент времени i. Любой сигнал $a(t)$ можно представить как линейную комбинацию задержанных импульсов:

$$a(t) = \sum_{i=0}^{T-1} a(i)\delta(t-i)$$

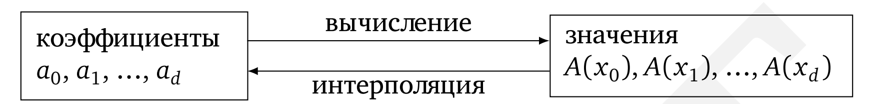
(если сигнал состоит из T отсчётов). По свойству линейности отклик на сигнал $a(t)$ определяется откликом на различные $\delta(t-i)$. А по свойству инвариантности эти отклики получаются сдвигом *импульсной характеристики* $b(t)$ (impulse response) — реакции системы на импульс $\delta(t)$.

Другими словами, отклик системы задаётся формулой

$$c(k) = \sum_{i=0}^{k} a(i)b(k-i),$$

а это в точности формула умножения многочленов!

Многочлены, однако, обычно даны нам в виде списка коэффициентов, и произведение необходимо вычислить в таком же виде. Поэтому умножение будет в три этапа: сначала мы перейдём от коэффициентов к значениям в точках, то есть *вычислим* значения многочлена в этих точках. После этого найдём значения произведения в точках и, наконец, произведём *интерполяцию* (interpolation), то есть восстановим коэффициенты произведения по его значениям.



Соответствующий алгоритм приведён на рис. 2.5. Ясно, что алгоритм корректен, но насколько он эффективен? Умножение значений требует линейного времени.[1] Но сколько времени нужно для вычисления значений (и интерполяции, о которой мы пока почти ничего не знаем)? Значение многочлена степени $d \leqslant n$ в точке можно вычислить за $O(d)$ операций (упражнение 2.29), поэтому значения многочлена степени n в n точках можно вычислить за $\Theta(n^2)$ операций. Быстрое преобразование Фурье решает эту задачу за время $O(n \log n)$ для специально выбранных точек $x_0, x_1, \ldots, x_{n-1}$; ускорение происходит за счёт того, что одни и те же вычисления оказываются полезны для разных точек.

Рис. 2.5. Алгоритм умножения многочленов.

{Вход: коэффициенты многочленов $A(x)$ и $B(x)$ степени не выше d.}
{Выход: коэффициенты многочлена $C = A \cdot B$.}

выбор
  выбрать точки $x_0, x_1, \ldots, x_{n-1}$ в количестве $n \geqslant 2d + 1$
вычисление
  вычислить $A(x_0), A(x_1), \ldots, A(x_{n-1})$ и $B(x_0), B(x_1), \ldots, B(x_{n-1})$
умножение
  вычислить $C(x_k) = A(x_k)B(x_k)$ для всех $k = 0, 1, \ldots, n-1$
интерполяция
  восстановить $C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$

2.6.2. Вычисление методом «разделяй и властвуй»

Допустим, что для вычисления значений многочлена $A(x)$ степени не выше $n-1$ мы выбрали n точек, разбитых на пары отличающихся знаком:

$$\pm x_0,\ \pm x_1,\ \ldots,\ \pm x_{n/2-1}.$$

[1] Мы считаем, что коэффициенты многочленов являются вещественными числами и что их размер позволяет производить с ними арифметические операции (сложение и умножение) за один шаг. Оценки, которые мы получим, легко обобщаются и на случай больших коэффициентов.

Ясно, что вычисления значений $A(x_i)$ и $A(-x_i)$ имеют много общего, поскольку чётные степени x_i и $-x_i$ совпадают.

Чтобы воспользоваться этим, выделим в $A(x)$ мономы чётных и нечётных степеней, скажем,

$$3+4x+6x^2+2x^3+x^4+10x^5=(3+6x^2+x^4)+x(4+2x^2+10x^4).$$

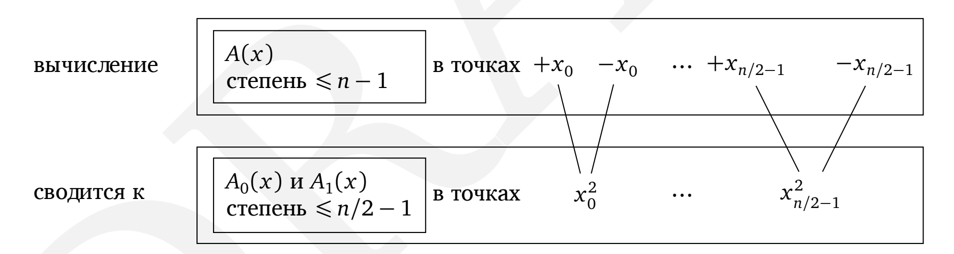
Выражения в скобках являются многочленами от x^2. В общем случае, многочлен $A(x)$ представляем как сумму

$$A(x)=A_0(x^2)+xA_1(x^2),$$

где коэффициенты многочленов A_0 и A_1 соответствуют коэффициентам A при чётных и нечётных степенях x. Степени многочленов A_0 и A_1 не больше $n/2-1$ (для удобства считаем, что n чётно). Теперь видно, что вычисление $A(x_i)$ и $A(-x_i)$ имеет много общего:

$$A(x_i)=A_0(x_i^2)+x_iA_1(x_i^2),$$
$$A(-x_i)=A_0(x_i^2)-x_iA_1(x_i^2).$$

Иначе говоря, вычисление значений $A(x)$ в n точках $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$ сводится к вычислению значений многочленов A_0 и A_1 (каждый из которых имеет вдвое меньшую степень) в $n/2$ точках $x_0^2, \ldots, x_{n/2-1}^2$.



Этим способом задача размера n сводится к двум задачам размера $n/2$, после чего за линейное время можно найти ответ. Если бы мы могли использовать эту идею рекурсивно, то получили бы соотношение

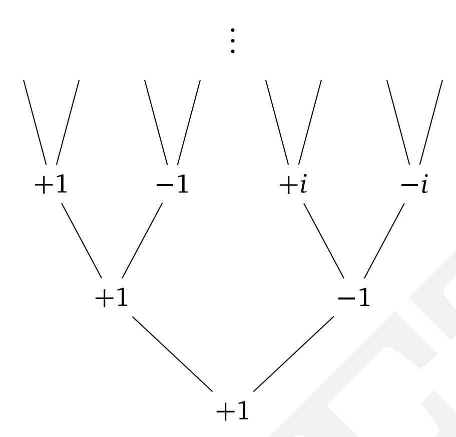
$$T(n)=2T\left(\frac{n}{2}\right)+O(n),$$

из которого бы следовала желаемая оценка $O(n\log n)$.

Проблема же здесь в том, что выбранные нами числа разбиты на пары лишь на первом шаге. Чтобы продолжить рекурсию, нам нужно, чтобы числа $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ тоже были разбиты на пары отличающихся знаком, но это невозможно, поскольку квадрат числа не может быть отрицательным.

Зато так может быть для комплексных чисел, в отличие от вещественных. Так почему бы не использовать комплексные числа? Но какие именно? Чтобы понять это, давайте посмотрим, чем должна закончиться рекурсия. На

последнем шаге у нас останется одна точка; пусть это будет 1. Тогда на предыдущем шаге были корни из 1, то есть ± 1:



На уровне выше тогда будут опять $\pm\sqrt{1} = \pm 1$, но теперь ещё и $\pm\sqrt{-1} = \pm i$, где i — комплексный корень из -1. Поднимаясь выше, мы должны прийти к изначальным n точкам. Наверное, вы уже догадались, какими будут эти точки: это *комплексные корни n-й степени из единицы* (complex nth roots of unity), то есть n решений уравнения $z^n = 1$. (Здесь n — степень двойки.)

На рис. 2.6 приведены основные факты о комплексных числах. В третьей сверху части рисунка показаны корни n-й степени из единицы, то есть числа $1, \omega, \omega^2, \ldots, \omega^{n-1}$, где $\omega = e^{2\pi i/n}$.

Для чётного n верны следующие утверждения:

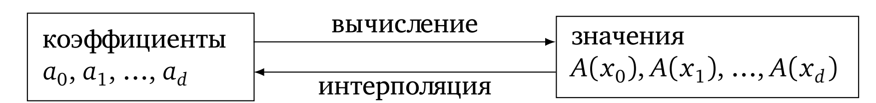
1. Корни n-й степени из единицы разбиваются на пары так, что числа в парах отличаются только знаком: $\omega^{n/2+j} = -\omega^j$.

2. Квадраты корней n-й степени — это корни степени $n/2$.

Таким образом, начав с корней степени n, где n есть степень двойки, мы последовательно будем получать подзадачи для корней степени $n/2^k$ для $k = 0, 1, 2, \ldots$ Как мы говорили, такие корни разбиваются на пары, что позволяет нам использовать метод «разделяй и властвуй». Соответствующий алгоритм называется *быстрым преобразованием Фурье* (fast Fourier transform, FFT, см. рис. 2.7).

2.6.3. Интерполяция

Вернёмся к нашей схеме умножения многочленов (рис. 2.5).



Если точки x_i являются корнями $(1, w, w^2, \ldots, w^{n-1})$ n-й степени из единицы, то при помощи быстрого преобразования Фурье можно перейти от коэф-

Рис. 2.6. Комплексные корни из 1 и рекурсивные вычисления.

мнимая ось
b
r
θ
a
вещественная ось

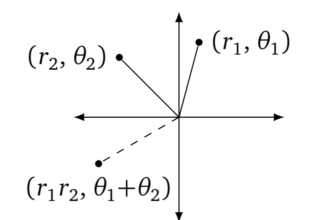
Комплексная плоскость

Числу $z = a + bi$ соответствует точка (a, b).
Полярные координаты:
$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, обозначение: (r, θ);

- *радиус* $r = \sqrt{a^2 + b^2}$;
- *угол* $\theta \in [0; 2\pi)$: $\cos\theta = a/r$, $\sin\theta = b/r$.

Примеры:

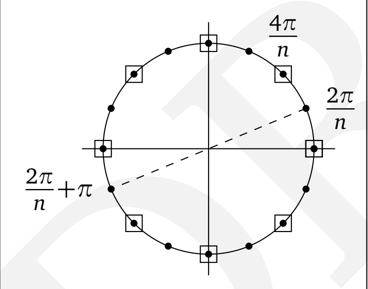
число	-1	i	$5+5i$
полярные координаты	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$



Умножение в полярных координатах

Умножаем радиусы и суммируем углы:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.
Для любого $z = (r, \theta)$

- $-z = (r, \theta + \pi)$, поскольку $-1 = (1, \pi)$;
- если z лежит на единичной окружности (то есть $r = 1$), то $z^n = (1, n\theta)$.

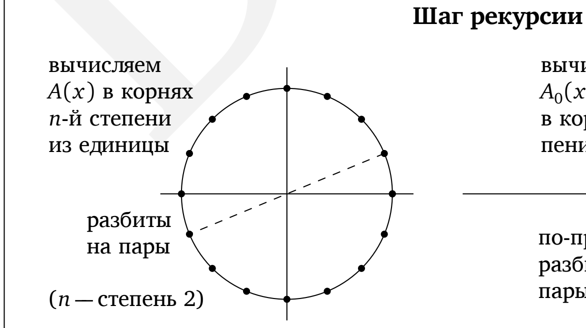


Комплексные корни степени n из 1

Корни уравнения $z^n = 1$.
По правилу умножения корнями являются числа $z = (1, 2\pi k/n)$, где $k = 0, 1, \ldots, n-1$ (показан случай $n = 16$).
Для чётных n

- корни разбиты на пары: $-(1, \theta) = (1, \theta + \pi)$;
- квадраты корней являются корнями из 1 степени $n/2$ (обведены в рамку).

Шаг рекурсии



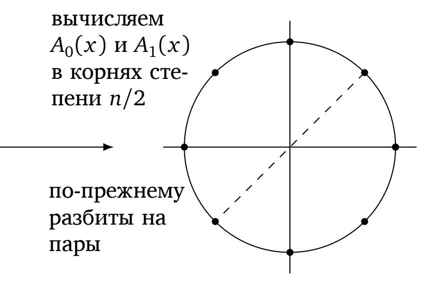


Рис. 2.7. Быстрое преобразование Фурье.

функция $\mathrm{FFT}(A, \omega)$

{Вход: коэффициенты многочлена $A(x)$ степени не более $n-1$, где
  n — степень двойки; комплексное число ω, для которого $\omega^n = 1$.}
{Выход: значения $A(\omega^0), \ldots, A(\omega^{n-1})$.}

если $\omega = 1$: вернуть $A(1)$
представить $A(x)$ в виде $A_0(x^2) + xA_1(x^2)$
вызвать $\mathrm{FFT}(A_0, \omega^2)$ для вычисления A_0 в чётных степенях ω
вызвать $\mathrm{FFT}(A_1, \omega^2)$ для вычисления A_0 в нечётных степенях ω
для j от 0 до $n-1$:
  $A(\omega^j) = A_0(\omega^{2j}) + \omega^j A_1(\omega^{2j})$
вернуть $A(\omega^0), \ldots, A(\omega^{n-1})$

фициентов многочлена к его значениям в этих точках за время $O(n \log n)$:

$$\langle\text{значения}\rangle = \mathrm{FFT}(\langle\text{коэффициенты}\rangle, \omega).$$

Но как перейти обратно (выполнить интерполяцию)? Поразительным образом оказывается, что

$$\langle\text{коэффициенты}\rangle = \frac{1}{n}\mathrm{FFT}(\langle\text{значения}\rangle, \omega^{-1}).$$

Обратный переход, таким образом, осуществляется с помощью всё того же быстрого преобразования Фурье, но с ω^{-1} вместо ω. На первый взгляд это кажется чудесным совпадением. Мы убедимся в верности этого факта, переформулировав операции с многочленами в терминах линейной алгебры. А пока отметим, что все шаги нашего алгоритма умножения многочленов за время $O(n \log n)$ (рис. 2.5) теперь полностью определены.

Формулировка в терминах матриц

Интересующее нас утверждение (интерполяция — это тоже преобразование Фурье) в конечном счёте сводится к несложному вычислению (по формуле для суммы геометрической прогрессии). Но полезно понимать более общий контекст в терминах линейной алгебры.

Мы имеем дело с двумя представлениями многочленов степени не выше $n-1$: в виде вектора коэффициентов и в виде вектора значений. Переход от коэффициентов к значениям — линейное преобразование, и матрицу этого преобразования легко написать:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Эта матрица называется *матрицей Вандермонда* (Vandermonde matrix).

Как мы уже говорили, коэффициенты многочлена степени не выше $n-1$ однозначно восстанавливаются по его значениям в n точках. В терминах линейной алгебры это соответствует такому свойству матрицы Вандермонда M:

> Если $x_0, \ldots, x_{n-1}$ попарно различны, то M обратима.

Задача интерполяции состоит в нахождении обратного преобразования, которое, как учит линейная алгебра, сводится к умножению на обратную матрицу:

> Вычисление значений есть умножение на M, а интерполяция — умножение на M^{-1}.

Задача интерполяции состоит в нахождении обратной матрицы, и нам нужно показать, что в интересующем нас случае матрицы Вандермонда $M(\omega)$ для точек $x_i = \omega^i$, где $\omega^n = 1$, имеет место следующее равенство.

Формула обращения.

$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1}).$$

Заметим, что ω^{-1} тоже будет корнем из единицы (сопряжённым к ω).

Матрица интерполяции

Как мы видели, преобразование Фурье умножает произвольный вектор размера n, содержащий коэффициенты многочлена, на $(n \times n)$-матрицу

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^j & \omega^{2j} & \ldots & \omega^{(n-1)j} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{matrix} \leftarrow & \text{строка для } \omega^0 = 1 \\ \leftarrow & \omega \\ \leftarrow & \omega^2 \\ & \vdots \\ \leftarrow & \omega^j \\ & \vdots \\ \leftarrow & \omega^{n-1} \end{matrix}$$

где ω — корень степени n из 1 (в качестве n мы берём степень двойки, но сейчас это не важно). Другими словами, (j, k)-й элемент $[M_n(\omega)]_{j,k}$ равен ω^{jk} (при нумерации строк и столбцов с нуля).

Нам осталось доказать такой факт:

Лемма. *$M_n(\omega)M_n(\omega^{-1}) = nI$, где I — единичная матрица.*

Доказательство. В самом деле, (j, k)-й элемент такого произведения по определению равен

$$\sum_{i=0}^{n-1} [M_n(\omega)]_{j,i} [M_n(\omega^{-1})]_{i,k} = \sum_{i=0}^{n-1} \omega^{ji} (\omega^{-1})^{ik} = 1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}.$$

Это сумма геометрической прогрессии со знаменателем ω^{j-k}. Если знаменатель равен 1, то есть $j = k$, то все члены суммы равны и вся сумма равна n. В противном случае по формуле суммы геометрической прогрессии получаем $(1-\omega^{n(j-k)})/(1-\omega^{(j-k)})$, что равно нулю, поскольку $\omega^n = 1$. □

В терминах линейной алгебры можно сказать немного иначе. Матрица $M_n(\omega^{-1})$ является эрмитово сопряжённой к матрице $M_n(\omega)$, то есть получается транспонированием (что в данном случае не меняет матрицы) и сопряжением всех её элементов по формуле $(a+bi)^* = a - bi$; в полярных координатах сопряжение комплексного числа соответствует изменению знака угла, то есть переходу от $z = re^{i\theta}$ к $z^* = re^{-i\theta}$. Для комплексного числа z единичной длины (в частности, для корней из единицы) $z^* = z^{-1}$. Наше равенство приобретает вид $MM^* = nI$, что означает, что столбцы матрицы M ортогональны относительно скалярного произведения (inner product)

$$u \cdot v^* = u_0 v_0^* + \ldots + u_{n-1} v_{n-1}^*$$

и имеют длину $\sqrt{n}$. Можно сказать ещё, что в пространстве многочленов степени не выше $n-1$ у нас есть два базиса: один — обычный, состоящий из одночленов, и другой — базис Фурье, в котором координаты многочлена — это его значения в корнях из единицы. Матрица $M_n(\omega)$ преобразует координаты из первого базиса во второй, и при этом она унитарна (с точностью до коэффициента), то есть эти два базиса задают одно и то же скалярное произведение в пространстве многочленов (опять же с точностью до коэффициента).

В базисе Фурье умножать многочлены легко, поэтому для умножения многочленов, заданных наборами коэффициентов, мы переходим к базису Фурье, после чего вычисляем произведение и результат переводим обратно в стандартный базис (интерполяция). Быстрое преобразование Фурье позволяет легко переходить от одного базиса к другому (в обе стороны).

2.6.4. Ещё о быстром преобразовании Фурье

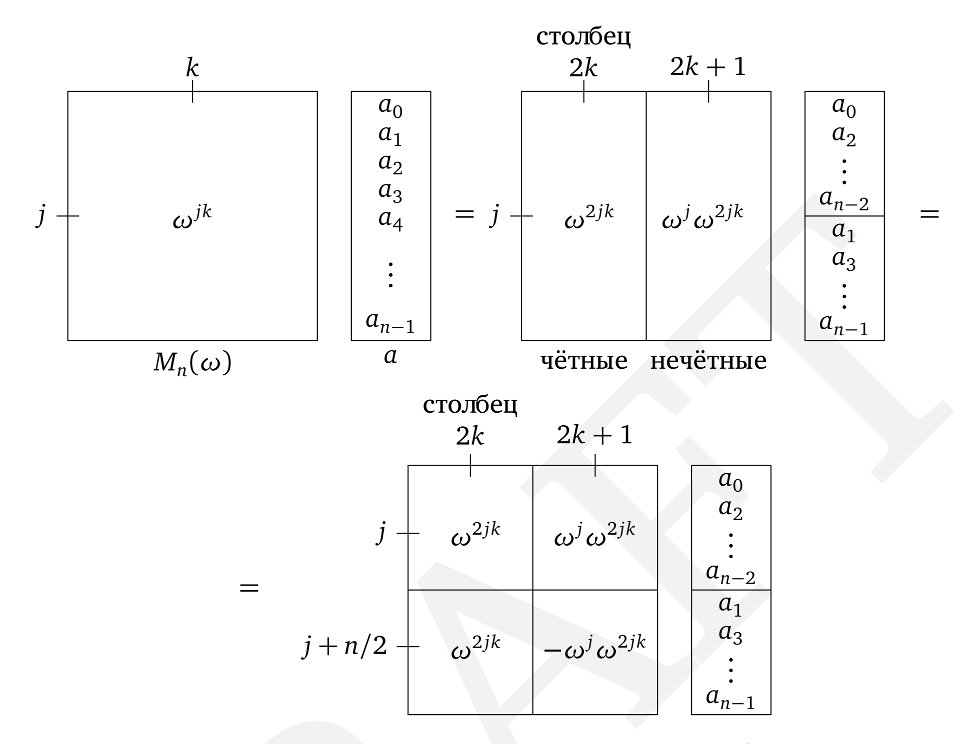
Итак, мы описали все шаги нашей схемы умножения многочленов. Рассмотрим теперь детали реализации самой важной части — быстрого преобразования Фурье.

Алгоритм быстрого преобразования Фурье

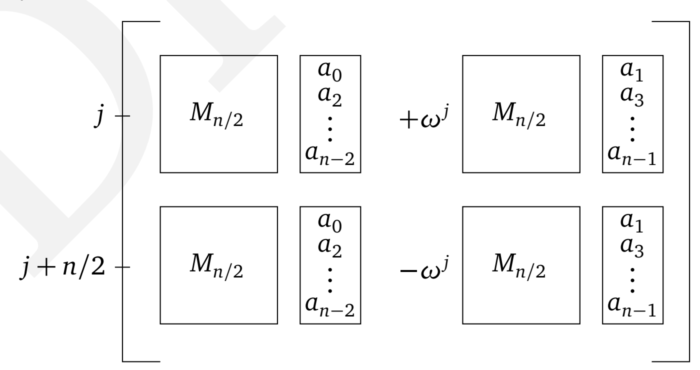
Входом БПФ является вектор $a = (a_0, \ldots, a_{n-1})$ и комплексное число ω; оно является первообразным корнем степени n из единицы, то есть его степени $1, \omega, \omega^2, \ldots, \omega^{n-1}$ различны и среди них есть все корни n-й степени из единицы. Алгоритм умножает вектор a на матрицу $M_n(\omega)$ размера $n \times n$, в которой (j, k)-й элемент (при нумерации строк и столбцов с нуля) равен ω^{jk}.

Возможность использования метода «разделяй и властвуй» становится яснее, если переставить координаты на входе (и соответствующие столбцы мат-

рицы), разделив их на чётные и нечётные:



Во втором равенстве мы воспользовались тем, что $\omega^{n/2} = -1$ и $\omega^n = 1$. Видно, что в левом верхнем и в левом нижнем углах теперь стоит матрица $M_{n/2}(\omega^2)$ размера $\frac{n}{2} \times \frac{n}{2}$. Справа же стоят матрицы, получающиеся из $M_{n/2}(\omega^2)$ домножением j-й строки на ω^j и $-\omega^j$ соответственно (для всех j). В итоге получаем



Мы свели вычисление произведения матрицы $M_n(\omega)$ на вектор $(a_0, \ldots, a_{n-1})$, то есть задачу размера n, к двум задачам размера $n/2$ — вычислению произведения $M_{n/2}(\omega^2)$ на $(a_0, a_2, \ldots, a_{n-2})$ и на $(a_1, a_3, \ldots, a_{n-1})$. Получился (рис. 2.8) тот же самый рекурсивный алгоритм быстрого преобразования Фурье, кото-

рый мы описывали раньше, только теперь он изложен в терминах матриц (вместо многочленов). Его время работы $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

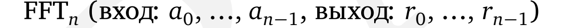
Рис. 2.8. Быстрое преобразование Фурье.

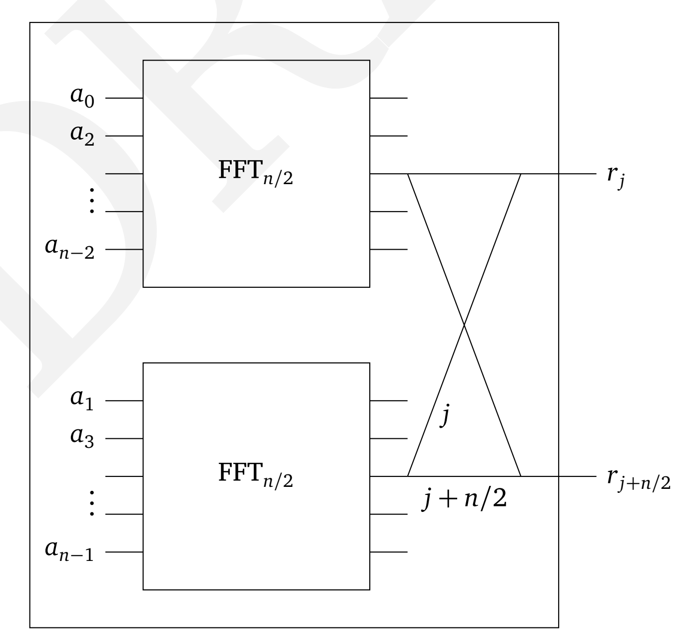
функция FFT(A, ω)
{Вход: вектор $a = (a_0, a_1, \ldots, a_{n-1})$, где n — степень двойки,
  ω — первообразный корень степени n из единицы.}
{Выход: значения $A(\omega^0), \ldots, A(\omega^{n-1})$.}

если $\omega = 1$: вернуть a
$(s_0, s_1, \ldots, s_{n/2-1}) \leftarrow$ FFT$((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
$(s'_0, s'_1, \ldots, s'_{n/2-1}) \leftarrow$ FFT$((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
для j от 0 до $n/2 - 1$:
  $r_j = s_j + \omega^j s'_j$
  $r_{j+n/2} = s_j - \omega^j s'_j$
вернуть $(r_0, r_1, \ldots, r_{n-1})$

Нерекурсивное описание БПФ

Покажем, как можно описать быстрое преобразование Фурье без рекурсии. Для начала изобразим один уровень рекурсии в виде схемы. На рисунке задача размера n сводится к двум задачам размера $n/2$.





Провода на рисунке передают комплексные числа слева направо. Пометка (*вес*) j на проводе означает, что соответствующее число домножается на ω^j. Когда два провода соединяются, соответствующие числа складываются. Таким образом, в двух выделенных на рисунке выходах происходят такие вычисления:

$$r_j = s_j + \omega^j s'_j,$$
$$r_{j+n/2} = s_j - \omega^j s'_j.$$

На рис. 2.9 показана вся схема при $n = 8$. (Места схемы, где происходит соединение проводов, показаны кружочками, остальные пересечения линий — кажущиеся.)

Отметим несколько фактов.

1. При n входах у схемы будет $\log_2 n$ уровней, на каждом уровне по n вершин. Всего в схеме производится $n \log n$ операций.

2. Входы даются в странном порядке: 0, 4, 2, 6, 1, 5, 3, 7.

Почему такой порядок? Как мы помним, в алгоритме быстрого преобразования Фурье сначала производится рекурсивный вызов отдельно для коэффициентов с чётными номерами (последний бит индекса равен нулю), а потом — с нечётными (последний бит равен единице). На следующем уровне рекурсии отдельно обрабатываются чётные коэффициенты из первой группы (00 на конце) и отдельно нечётные (10 на конце) и так далее. Другими словами, индексы упорядочены по последнему биту, в случае равенства — по предпоследнему и так далее (как в обычной двоичной записи, только спра-

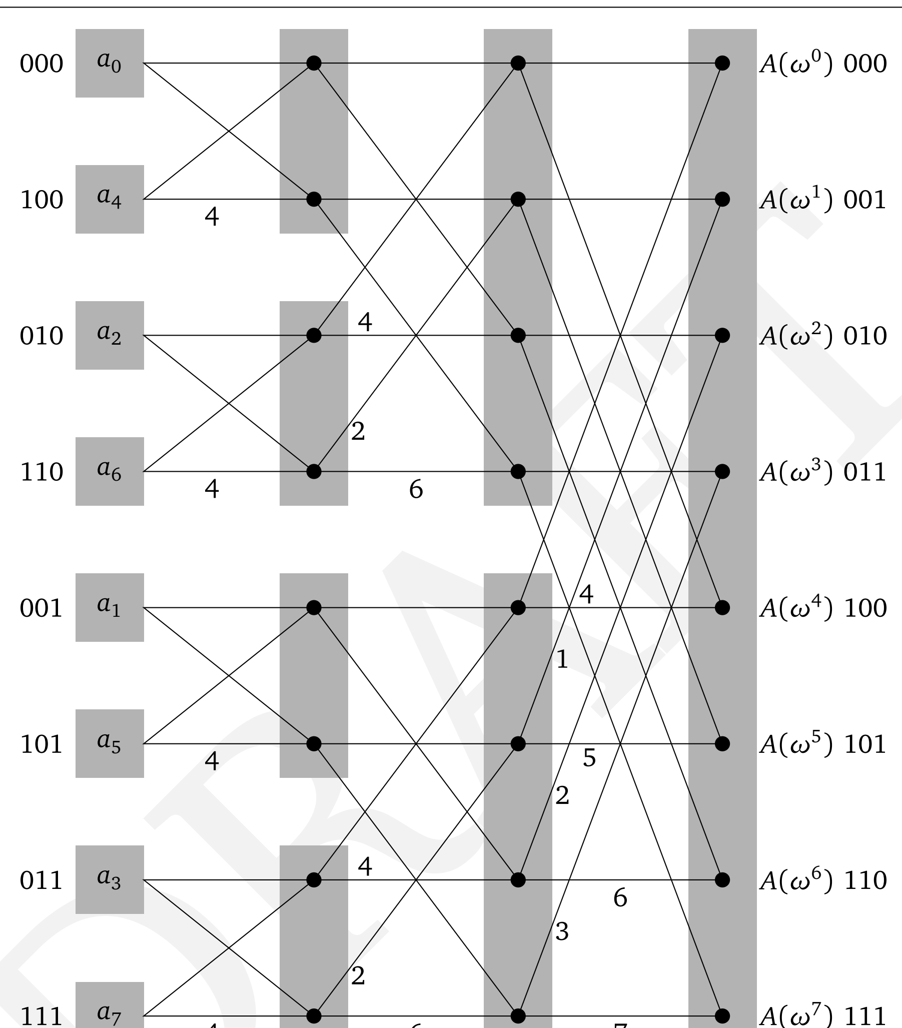
Медленное распространение быстрого алгоритма

На заседании научного совета при президенте Кеннеди в 1963 году математик из Принстона, Джон Тьюки, объяснил Дику Гарвину из компании IBM, как быстро посчитать преобразование Фурье. В то время Гарвин работал над распознаванием ядерных взрывов по сейсмографическим данным, и преобразование Фурье было узким местом. Вернувшись с заседания, Гарвин попросил Джона Кули реализовать алгоритм Тьюки.

Гарвин и Кули решили, что надо опубликовать алгоритм, чтобы эту идею кто-нибудь не запатентовал. Сам Тьюки медлил, поэтому этим занялся Кули, и в 1965 году появилась их совместная с Тьюки статья, ставшая одной из наиболее известных и цитируемых. Тьюки не торопился с публикацией не потому, что хотел сохранить секрет или поживиться на патенте. Он считал, что метод достаточно прост и скорее всего уже известен. Да и вообще в то время алгоритмы мало ценили.

Но на самом деле Тьюки был прав: как позже выяснилось, британские инженеры использовали БПФ уже в 1930-е годы. Более того, схожие идеи использовались в статье Гаусса, которого мы уже вспоминали в начале этой главы. Эта его работа долгое время оставалась неизвестной для современных учёных, потому что была написана на латыни (как это тогда было принято).

Рис. 2.9. Схема быстрого преобразования Фурье.



ва налево): надо взять обычную запись $000, 001, 010, \ldots$ и перевернуть все числа.

3. Для каждого входа a_j и каждого выхода $A(\omega^k)$ существует ровно один соединяющий их путь.

Более того, этот путь довольно просто описать, используя двоичное представление индексов. Из каждой вершины выходит вправо ровно два ребра: верхнее (0-ребро) и нижнее (1-ребро). Чтобы попасть в $A(\omega^k)$ из любого входа, нужно читать двоичную запись k справа налево и идти по соответствующим рёбрам слева направо. (Опишите подобным образом и обратный путь.)

4. На пути из a_j в $A(\omega^k)$ сумма весов равна $jk \bmod 8$.

Так как $\omega^8 = 1$, вклад входа a_j в $A(\omega^k)$ есть в точности $a_j\omega^{jk}$, и поэтому схема корректно вычисляет значения на выходах.

5. Наконец, отметим, что такое представление схемы очень удобно для параллельного вычисления и аппаратной реализации.

Упражнения

2.1. Найдите произведение чисел 10011011 и 10111010 (в двоичной системе счисления), используя алгоритм умножения чисел, основанный на методе «разделяй и властвуй».

2.2. Покажите, что для любых целых положительных n и b на отрезке $[n, bn]$ обязательно найдётся степень b.

2.3. В разделе 2.2 мы анализировали рекуррентные соотношения, оценивая число операций на каждом уровне дерева рекурсии. Примерно тот же анализ можно изложить иначе. Для примера рассмотрим знакомое нам соотношение $T(n) = 2T(n/2) + O(n)$ и будем его «раскручивать». Первым делом заменим $O(n)$ на верхнюю оценку cn для некоторой константы c, получив $T(n) \leqslant 2T(n/2) + cn$. Применяя это неравенство несколько раз, мы оцениваем $T(n)$ сначала через $T(n/2)$, потом через $T(n/4)$, через $T(n/8)$ и так далее. В конце концов мы дойдём до $T(1) = O(1)$:

$$\begin{aligned} T(n) &\leqslant 2T\left(\frac{n}{2}\right) + cn \leqslant \\ &\leqslant 2\left[2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn = 4T\left(\frac{n}{4}\right) + 2cn \leqslant \\ &\leqslant 4\left[2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right] + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \leqslant \\ &\leqslant 8\left[2T\left(\frac{n}{16}\right) + \frac{cn}{8}\right] + 3cn = 16T\left(\frac{n}{16}\right) + 4cn \leqslant \ldots \end{aligned}$$

Уже виден и общий вид такого неравенства:

$$T(n) \leqslant 2^k T\left(\frac{n}{2^k}\right) + kcn.$$

Подставляя $k = \log_2 n$, получаем $T(n) \leqslant nT(1) + cn\log_2 n = O(n\log n)$.

(a) Проведите те же рассуждения для $T(n) = 3T(n/2) + O(n)$. Каков будет общий вид неравенства в данном случае? Какое k нужно подставить?

(b) Теперь оцените $T(n)$, если $T(n) = T(n-1) + O(1)$ (это соотношение не покрывается нашей теоремой).

2.4. Какой из трёх приведённых ниже алгоритмов вы бы предпочли? Каково время работы этих алгоритмов?

- Алгоритм A, решая задачу, производит пять рекурсивных вызовов для подзадач вдвое меньшего размера, после чего строит ответ для исходной задачи за линейное время.

- Алгоритм B, решая задачу размера n, делает два рекурсивных вызова для задач размера $n-1$, после чего находит ответ за время $O(1)$.
- Алгоритм C, решая задачу размера n, рекурсивно решает девять задач размера $n/3$ и строит ответ за время $O(n^2)$.

2.5. Найдите решение для каждого из приведённых ниже рекуррентных соотношений в терминах Θ:

(a) $T(n) = 2T(n/3) + 1$;

(b) $T(n) = 5T(n/4) + n$;

(c) $T(n) = 7T(n/7) + n$;

(d) $T(n) = 9T(n/3) + n^2$;

(e) $T(n) = 8T(n/2) + n^3$;

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$;

(g) $T(n) = T(n-1) + 2$;

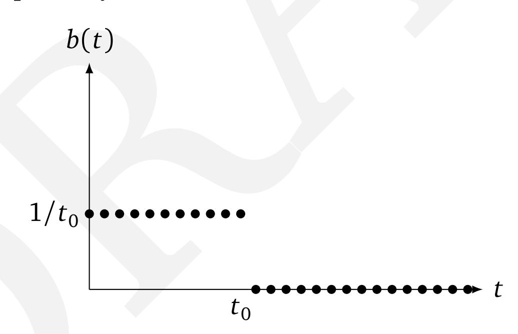
(h) $T(n) = T(n-1) + n^c$, где $c \geqslant 1$ — константа;

(i) $T(n) = T(n-1) + c^n$, где $c > 1$ — константа;

(j) $T(n) = 2T(n-1) + 1$;

(k) $T(n) = T(\sqrt{n}) + 1$.

2.6. Линейная и инвариантная по времени система имеет следующую импульсную характеристику:



(a) Опишите словами работу данной системы.

(b) Умножение на какой многочлен ей соответствует?

2.7. Чему равна сумма всех корней n-й степени из единицы? Чему равно их произведение, если n чётно? Если n нечётно?

2.8. (a) Примените алгоритм БПФ к вектору $(1, 0, 0, 0)$. Каково соответствующее ω? У какого вектора БПФ равно $(1, 0, 0, 0)$?

(b) Те же вопросы для вектора $(1, 0, 1, -1)$.

2.9. *Умножение многочленов при помощи БПФ.*

(a) Допустим, мы хотим найти произведение многочленов $x+1$ и x^2+1 при помощи быстрого преобразования Фурье. Выберите подходящую сте-

пень двойки, примените БПФ к двум векторам, перемножьте полученные векторы покомпонентно и получите искомый многочлен обратным БПФ.

(b) Повторите это для многочленов $1 + x + 2x^2$ и $2 + 3x$.

2.10. Найдите коэффициенты многочлена p степени не больше 4, если $p(1) = 2$, $p(2) = 1$, $p(3) = 0$, $p(4) = 1$, $p(5) = 0$.

2.11. В алгоритме для умножения матриц (раздел 2.5) мы использовали следующее свойство: если X и Y суть матрицы размера $n \times n$ и

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

где A, B, C, D, E, F, G и H являются матрицами $\frac{n}{2} \times \frac{n}{2}$, то произведение XY может быть записано следующим образом:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Докажите это.

2.12. Оцените порядок количества строк, которое напечатает приведённая ниже программа, как функции от n: запишите рекуррентное соотношение и решите его. Можно считать, что n есть степень двойки.

```
процедура F(n)
если n > 1:
  напечатать «всё ещё работаю»
  F(n/2)
  F(n/2)
```

2.13. Двоичное дерево называется *строго двоичным*, если у каждой его вершины либо ноль, либо два сына. Обозначим через B_n количество всех строго двоичных деревьев с n вершинами.

(a) Нарисуйте все строго двоичные деревья с 3, 5, 7 вершинами и найдите значения B_3, B_5 и B_7. Почему мы пропустили чётные числа (например, B_4)?

(b) Запишите рекуррентное соотношение для B_n.

(c) Докажите индукцией по n, что $B_n = \Omega(2^{n/2})$.

2.14. Покажите, как за время $O(n \log n)$ удалить из массива размера n все дубликаты, то есть оставить каждый элемент в одном экземпляре.

2.15. Основной составляющей алгоритма нахождения медианы (раздел 2.4) является процедура SPLIT, которая получает на вход массив S и элемент v и делит элементы массива на три части: меньшие v, равные v и большие v. Покажите, как сделать это на том же месте (in place), то есть не выделяя памяти под новый массив.

2.16. Дан бесконечный массив $A[\cdot]$, первые n элементов которого содержат возрастающую последовательность целых чисел, а дальше стоит ∞. Значение n *не* дано. Постройте алгоритм, который получает на вход целое число x

и находит это x в массиве — или говорит, что там его нет, за время $O(\log n)$. (Если вас смущает тот факт, что массив бесконечен, можно представлять себе массив неизвестной длины n и функцию, которая при обращении к элементу с номером $i > n$ возвращает ∞.)

2.17. В массиве $A[1\ldots n]$ записана возрастающая последовательность целых чисел. Постройте алгоритм типа «разделяй и властвуй», который проверяет, существует ли такой индекс i, что $A[i] = i$.

2.18. Рассмотрим задачу проверки принадлежности элемента x заданному упорядоченному массиву $A[1\ldots n]$. Такая задача решается двоичным поиском за время $O(\log n)$. Покажите, что любой алгоритм, решающий эту задачу, в некоторых случаях читает $\Omega(\log n)$ элементов этого массива.

2.19. *Слияние k массивов.* Допустим, у нас есть k упорядоченных массивов размера n и мы хотим получить из них один упорядоченный массив размера kn.

(a) Сразу приходит в голову следующая идея: при помощи процедуры Merge из раздела 2.3 сольём сначала первые два массива, результат сольём с третьим, потом с четвёртым и так далее. Каково время работы такого алгоритма (как функция от k и n)?

(b) Постройте более эффективный алгоритм, используя метод «разделяй и властвуй».

2.20. Покажите, как упорядочить массив целых чисел $x[1\ldots n]$ за время $O(n+M)$, где $M = \max_i x_i - \min_i x_i$. Для небольших M это даёт линейное время. Почему в данном случае неприменима нижняя оценка $\Omega(n \log n)$?

2.21. *Среднее арифметическое и медиана.* В статистике иногда нужно описать множество наблюдаемых значений $\{x_1, x_2, \ldots, x_n\} \subseteq \mathbb{R}$ одним числом. Часто в качестве такого числа выбирается медиана μ_1 или среднее арифметическое μ_2.

(a) Покажите, что медиана является значением параметра μ, которое минимизирует сумму

$$\sum_i |x_i - \mu|.$$

Для простоты можно считать, что n нечётно. (Подсказка: Как меняется эта сумма с изменением μ? когда она растёт, а когда убывает?)

(b) Покажите, что среднее арифметическое минимизирует функцию

$$\sum_i (x_i - \mu)^2.$$

Доказать это можно, найдя минимум этой функции (квадратного трёхчлена) или преобразовав её к виду

$$\sum_i (x_i - \mu)^2 = \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2.$$

Видно, что среднее арифметическое μ_2 «штрафует» точки, далёкие от μ, гораздо сильнее, чем μ_1. То есть μ_2 в большей степени заботится о далёких наблюдаемых значениях. В результате всего несколько наблюдений, сильно отклоняющихся от среднего, могут сильно изменить μ_2. Поэтому μ_1 иногда считается более разумным статистическим описанием набора наблюдений, чем μ_2. Другая крайность — это μ_∞, определяемое как значение μ, минимизирующее выражение

$$\max_i |x_i - \mu|.$$

(c) Покажите, что μ_∞ может быть вычислено за время $O(n)$ (считаем, что x_i достаточно малы, так что основные арифметические операции с ними производятся за один шаг).

2.22. Даны два упорядоченных массива размера m и n. Постройте алгоритм, находящий k-й по величине элемент в объединении этих массивов за время $O(\log(m+n))$.

2.23. *Представителем большинства* (majority element) в массиве $A[1\ldots n]$ называется элемент, равные которому составляют в массиве большинство. Постройте алгоритм, находящий такой элемент (если он есть; если нет, алгоритм может делать что угодно). На элементах массива не задан порядок, поэтому запросы типа «$A[i] > A[j]$?» запрещены. (Представьте себе, например, массив картинок.) Однако разрешены запросы типа «$A[i] = A[j]$?», ответы на которые даются за время $O(1)$.

(a) Покажите, как решить такую задачу за время $O(n \log n)$. (Подсказка: разбейте массив на две половины. Если бы вы знали представителей большинства в обеих половинах, могли бы вы быстро найти представителя большинства в исходном массиве? Если да, то можно воспользоваться методом «разделяй и властвуй».)

(b) Теперь постройте линейный алгоритм. (Подсказка: опишем другой подход, основанный на методе «разделяй и властвуй». Разобьём элементы массива на пары произвольным образом. Для каждой из $n/2$ пар: если оба элемента равны, то выкинем один из них, если же различны, то выкинем оба. Покажите, что после такой операции останется не больше $n/2$ элементов и представитель большинства сохранится, если он был. Впрочем, то же наблюдение можно использовать и более простым образом, поддерживая инвариант «искомый элемент является представителем большинства в массиве $A[k\ldots n]$, дополненном p копиями элемента $A[q]$» и увеличивая k.)

2.24. На с. 59 приведена общая идея алгоритма быстрой сортировки.

(a) Напишите псевдокод этого алгоритма.

(b) Докажите, что время работы алгоритма в *худшем случае* на массиве размера n есть $\Theta(n^2)$.

(c) Докажите, что математическое ожидание времени работы алгоритма (усреднение по всем вариантам выбора элементов для сравнения) на любом

входе размера n удовлетворяет рекуррентному соотношению

$$T(n) \leqslant O(n) + \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i))$$

и что решением данного соотношения является $O(n \log n)$.

2.25. В разделе 2.1 приведён алгоритм, перемножающий два n-битовых числа x и y за время $O(n^a)$, где $a = \log_2 3$. Назовём данный алгоритм FastMultiply(x, y).

(a) Пусть мы хотим найти двоичное представление десятичного числа 10^n (единица и n нулей). Будем считать, что n — степень двойки. Рассмотрим такой алгоритм:

```
функция Pwr2Bin(n)
если n = 1:  вернуть 1010₂
иначе:
  z = ...
  вернуть FastMultiply(z, z)
```

Впишите недостающую строчку псевдокода. Запишите рекуррентное соотношение на время работы алгоритма и решите его.

(b) Пусть теперь мы хотим перевести произвольное десятичное число x, в записи которого n цифр (n — степень двойки), в двоичную систему счисления. Алгоритм приведён ниже.

```
функция Dec2Bin(x)
если n = 1:  вернуть binary[x]
иначе:
  разделить x на два десятичных числа x_L, x_R из n/2 цифр
  вернуть ...
```

Массив binary[·] содержит заранее вычисленные двоичные представления десятичных цифр; доступ к любому элементу этого массива требует времени $O(1)$.

Как и в предыдущем упражнении, допишите псевдокод, запишите рекуррентное соотношение и решите его.

2.26. Профессор О. П. Рометчивый утверждает, что возведение n-битового числа в квадрат проще, чем умножение двух n-битовых чисел (в том смысле, что для возведения в квадрат есть асимптотически более быстрый алгоритм). Стоит ли ему верить?

2.27. *Квадратом* (square) матрицы A называется произведение её на себя, то есть AA.

(a) Покажите, что для вычисления квадрата матрицы размера 2×2 достаточно пяти умножений чисел.

(b) Найдите ошибку в приведённом ниже алгоритме вычисления квадрата матрицы размера $n \times n$:

> Действуем как в алгоритме Штрассена, но теперь вместо 7 рекурсивных вызовов нужно лишь 5, поскольку мы перемножаем не две различные матрицы, а две одинаковые (см. предыдущий пункт); из основной теоремы следует, что его время работы будет $O(n^{\log_2 5})$.

(c) На самом деле, вычисление квадрата матрицы и не может быть легче, чем вычисление произведения: если $(n \times n)$-матрицы можно возводить в квадрат за время $S(n) = O(n^c)$, то и произвольные две $(n \times n)$-матрицы можно перемножить за время $O(n^c)$. Покажите это, рассмотрев квадрат матрицы $\begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$, где X и Y — матрицы размера $n \times n$.

2.28. *Матрицы Адамара* (Hadamard matrices) $H_0, H_1, H_2, \ldots$ определяются следующим образом:

- H_0 — матрица размера 1×1, состоящая из одной единицы.
- При $k > 0$ матрица H_k имеет размер $2^k \times 2^k$ и определяется как

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right].$$

Покажите, что для вектор-столбца v длины $n = 2^k$ произведение $H_k v$ может быть вычислено за время $O(n \log n)$. Считаем, что размер используемых чисел позволяет производить основные арифметические операции с ними за один шаг.

2.29. Мы хотим вычислить значение многочлена $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots$ $\ldots + a_n x^n$ в точке x.

(a) Покажите, что следующий алгоритм, известный как *схема Горнера* (Horner's rule), правильно вычисляет значение $z = p(x)$:

```
z ← a_n
для i от n − 1 до 0:
  z ← zx + a_i
вернуть z
```

(b) Сколько арифметических операций производит данный алгоритм? Можете ли вы указать многочлен, для которого есть более быстрый способ вычисления значения в произвольной точке?

2.30. В данном упражнении показывается, как вычислять преобразование Фурье (ПФ) в арифметике сравнений, например, по модулю 7.

(a) Существует такое ω, что все степени $\omega, \omega^2, \ldots, \omega^6$ различны (по модулю 7). Найдите такое ω и покажите, что $\omega + \omega^2 + \ldots + \omega^6 = 0$. (Отметим также, что такое число существует для любого простого модуля.)

(b) Найдите преобразование Фурье вектора $(0, 1, 1, 1, 5, 2)$ по модулю 7, используя матричное представление, то есть умножьте данный вектор на $M_6(\omega)$ (для найденного ранее ω). Все промежуточные вычисления производите по модулю 7.

(c) Запишите матрицу обратного преобразования Фурье. Покажите, что при умножении на эту матрицу получается исходный вектор. (Как и прежде, все вычисления должны производиться по модулю 7.)

(d) Перемножьте многочлены $x^2 + x + 1$ и $x^3 + 2x - 1$ при помощи ПФ по модулю 7.

2.31. В разделе 1.2.3 мы познакомились с алгоритмом Евклида для вычисления наибольшего общего делителя (НОД) двух положительных целых чисел. Рассмотрим алгоритм вычисления НОД, основанный на методе «разделяй и властвуй».

(a) Докажите, что

$$\text{НОД}(a, b) = \begin{cases} 2\,\text{НОД}(a/2, b/2), & \text{если } a \text{ и } b \text{ чётны,} \\ \text{НОД}(a, b/2), & \text{если } a \text{ нечётно, а } b \text{ чётно,} \\ \text{НОД}((a-b)/2, b), & \text{если } a \text{ и } b \text{ нечётны.} \end{cases}$$

(b) Постройте алгоритм, основанный на этом соотношении.

(c) Какой алгоритм будет быстрее для n-битовых чисел a и b при больших значениях n — ваш или алгоритм Евклида? (Поскольку n велико, нельзя считать, что основные арифметические операции с числами производятся за один шаг.)

2.32. В данном упражнении мы построим алгоритм для геометрической задачи о *паре ближайших точек*.

Вход: список точек на плоскости

$$\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}.$$

Выход: пара ближайших точек, то есть такая пара точек $p_i \neq p_j$, расстояние между которыми, вычисляемое как

$$|p_i - p_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

минимально (среди всех пар).

Будем для простоты считать, что n есть степень двойки и что все абсциссы и ординаты данных точек различны.

Ниже приведены основные идеи алгоритма.

- Найдём такое x, что ровно для половины точек $x_i < x$ и ровно для половины точек $x_i > x$. Разобьём точки на соответствующие два множества L и R.
- Рекурсивно найдём пары ближайших точек отдельно в L и R. Назовём найденные пары $p_L, q_L \in L$ и $p_R, q_R \in R$, а расстояния между ними d_L и d_R соответственно. Пусть $d = \min\{d_L, d_R\}$.

- Остаётся проверить, есть ли в L точка, расстояние от которой до некоторой точки из R меньше d. Для этого удалим из рассмотрения все точки, для которых $|x_i - x| > d$, и упорядочим оставшиеся точки по значениям y.
- Теперь пройдёмся по упорядоченному списку и для каждой точки вычислим расстояние от неё до *семи* следующих за ней точек. Пару таких точек с минимальным расстоянием назовём $\{p_M, q_M\}$.
- Ответом считаем наилучшую из пар $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$.

(a) Покажите, что любой квадрат размера $d \times d$ на плоскости содержит не более четырёх точек из L.

(b) Докажите корректность алгоритма. (Сложен лишь случай, когда одна из точек искомой пары попала в L, а другая — в R.)

(c) Напишите псевдокод алгоритма и покажите, что время его работы удовлетворяет соотношению

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n).$$

Докажите, что решением соотношения является $O(n \log^2 n)$.

(d) Как улучшить алгоритм, сократив время работы до $O(n \log n)$?

2.33. Даны три $(n \times n)$-матрицы A, B, C. Мы хотим проверить равенство $AB = C$. Это можно сделать за время $O(n^{\log_2 7})$, воспользовавшись алгоритмом Штрассена. Но есть и более быстрый вероятностный алгоритм для такой проверки.

(a) Пусть $\mathbf{v}$ — случайный n-мерный вектор, каждая компонента которого выбирается равновероятно и независимо из множества $\{0, 1\}$. Докажите, что для любой ненулевой $(n \times n)$-матрицы M вероятность того, что $M\mathbf{v} = 0$, не больше $1/2$.

(b) Покажите, что $\mathrm{P}(AB\mathbf{v} = C\mathbf{v}) \leqslant 1/2$ при $AB \neq C$. Как использовать это для вероятностной проверки равенства $AB = C$ за время $O(n^2)$?

2.34. *Линейная* 3-*выполнимость*. Задача 3-выполнимости описана в разделе 8.1: дан набор дизъюнктов с тремя переменными и надо узнать, могут ли они одновременно быть истинными для какого-то набора значений `True`/`False`.

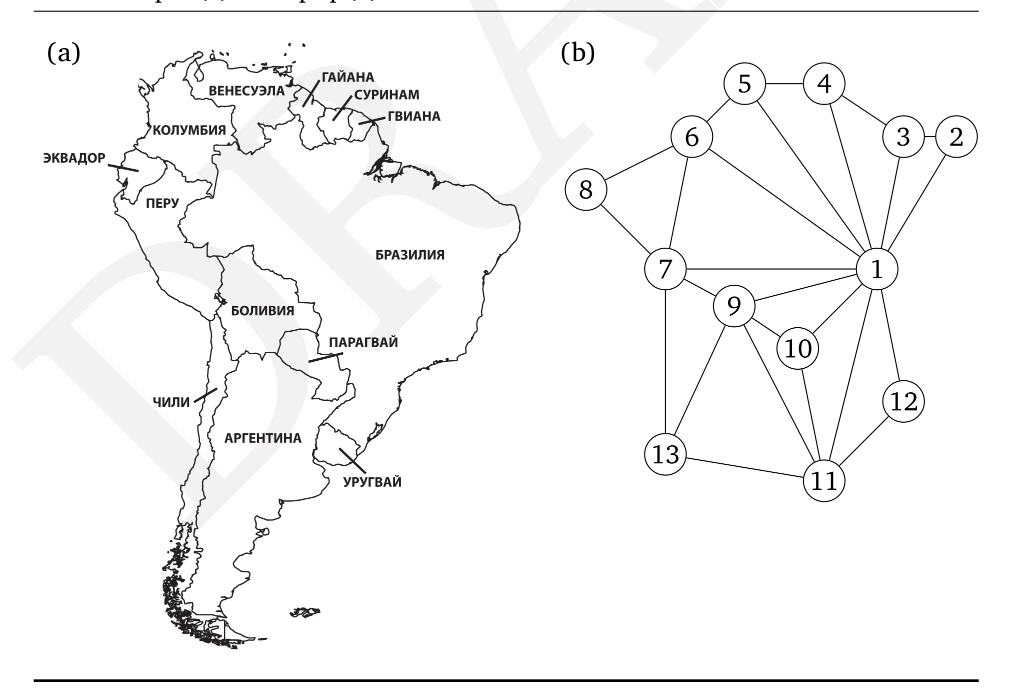
Пусть дополнительно известно, что формула в 3-КНФ с n переменными обладает таким свойством «локальности»: если переменные с номерами i и j входят в один дизъюнкт, то $|j - i| \leqslant 10$. Постройте линейный по времени алгоритм проверки выполнимости для таких формул.

Глава 3
Декомпозиция графов

3.1. Откуда берутся графы

Многие важные задачи легко формулируются на языке графов. Пусть, например, мы хотим выбрать цвет для каждой страны на карте, при этом соседи должны иметь разные цвета (иначе не будет видна их общая граница). Какое минимальное число цветов для этого нужно? Видно, что карта (рис. 3.1(a)) содержит много лишнего: моря, реки и т. п. Достаточно построить граф (рис. 3.1(b)), изобразив каждую страну *вершиной* (в нашем примере 1 — Бразилия, 11 — Аргентина и т. д.) и соединив соседей *ребром*. Надо покрасить все вершины графа в минимальное число цветов, при этом концы каждого ребра должны быть разного цвета.

Рис. 3.1. Карта (a) и её граф (b).



Одна и та же математическая формулировка возникает в разных практических задачах. Пусть, например, мы хотим назначить даты экзаменов так,

чтобы никому не надо было сдавать два экзамена в один день, и при этом хотим занять поменьше дней. Заведём вершину для каждого экзамена и соединим две вершины ребром, если хотя бы один студент сдаёт оба экзамена. Раскрасив граф в минимальное число цветов, получим оптимальное расписание (экзамены одного цвета назначаются на один день).

В этой главе мы рассмотрим некоторые базовые алгоритмы, связанные с разбиением графа на части. Начнём с определений.

Граф задаётся множеством вершин V и множеством рёбер E, соединяющих пары вершин. (Английская терминология: вершина называется vertex или node, а ребро — edge.)

В примере с картой $V = \{1, 2, 3, \ldots, 13\}$, а рёбра из E соединяют страны, граничащие друг с другом (в частности, E содержит рёбра $\{1, 2\}$, $\{9, 11\}$ и $\{7, 13\}$). Отношение «быть соседом» симметрично, так что ребро из x в y возникает одновременно с ребром из y в x. Поэтому мы записываем ребро, соединяющее x с y, как множество: $\{x, y\}$. Такие рёбра называются *неориентированными* (undirected), а соответствующий граф — *неориентированным графом* (undirected graph).

Бывают и несимметричные отношения, и их изображают *ориентированными рёбрами* (directed edges). В ориентированном графе наличие ребра из x в y не гарантирует наличие ребра из y в x. Ориентированное ребро из x в y будем обозначать (x, y). Можно считать «всемирную паутину» (World Wide Web) ориентированным графом: из вершины u идёт ориентированное ребро в вершину v, если на странице u есть ссылка на страницу v. В этом графе миллиарды вершин и рёбер, так что алгоритмы, с ним работающие, должны быть быстрыми (к счастью, такие алгоритмы есть — многое можно сделать за линейное время).

3.1.1. Представление графа

Рассмотрим граф с $n = |V|$ вершинами $v_1, \ldots, v_n$. Его *матрицей смежности* (adjacency matrix) называется $(n \times n)$-матрица a, в которой

$$a_{ij} = \begin{cases} 1, & \text{если есть ребро из } v_i \text{ в } v_j, \\ 0 & \text{в противном случае.} \end{cases}$$

Матрица смежности неориентированного графа, таким образом, симметрична. В таком представлении мы можем за время $O(1)$ проверить, соединены ли данные вершины ребром (посмотрев на один элемент массива). В то же время хранение матрицы требует памяти $O(n^2)$, что во многих случаях неэкономно. Альтернативное представление — *список смежности* (adjacency list). Требуемая при этом память пропорциональна размеру графа (сумме числа вершин и числа рёбер). Элементами списка смежности являются связные списки, по одному для каждой вершины графа. Для вершины u в таком списке хранятся вершины, в которые ведут рёбра из u, то есть вершины v, для которых $(u, v) \in E$. Для ориентированного графа каждое ребро входит только в один из этих списков (для начальной вершины), а для неориентированного в два (для двух концов ребра). Список смежности, таким образом, тре-

Матрица смежности или список смежности?

Что лучше? Ответ зависит от отношения между числом вершин $|V|$ и числом рёбер $|E|$. Заметим, что $|E|$ может быть довольно малым — порядка $|V|$ (если $|E|$ сильно меньше $|V|$, то граф уже вырожденный — в частности, содержит изолированные вершины). Или же довольно больши́м — порядка $|V|^2$ (если граф содержит все возможные рёбра). Графы с большим числом рёбер называют *плотными* (dense), с малым — *разреженными* (sparse).

Выбирая алгоритм, полезно понимать, с какими графами ему в основном придётся иметь дело. Важно это и при хранении: если хранить веб-граф, в котором больше восьми миллиардов вершин, в виде матрицы смежности, то занимать она будет *миллионы терабайтов*, что сравнимо с общей ёмкостью всех жёстких дисков в мире. И дальше, скорее всего, будет только хуже (матрица может расти быстрее, чем производство дисков).

А вот хранить граф Интернета в виде списка смежности вполне разумно, поскольку хранить нужно несколько десятков миллиардов гиперссылок, каждая из которых будет занимать в списке всего несколько байтов, и такого размера диск поместится в карман. Такая разница происходит из-за того, что граф Интернета очень разрежен: страница содержит в среднем пару десятков ссылок на другие страницы — из нескольких миллиардов возможных.

бует памяти $O(|V|+|E|)$. Проверка наличия ребра (u, v) теперь требует просмотра списка вершины u (что может быть больше $O(1)$ шагов). Зато в этом представлении легко просмотреть всех соседей заданной вершины (что часто бывает необходимо). Список смежности для неориентированного графа симметричен (если u содержится в списке для v, то и v содержится в списке для u).

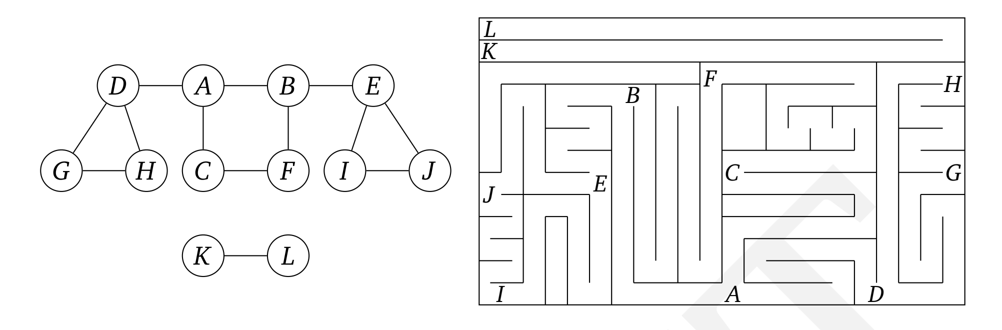
3.2. Поиск в глубину в неориентированных графах

3.2.1. Прохождение лабиринтов

Поиск в глубину (depth-first search) отвечает на вопрос: какие части графа достижимы из заданной вершины? Это похоже на задачу обхода лабиринта: имея граф как список смежности, мы в каждой вершине «видим» её соседей — как в лабиринте, где мы видим соседние помещения (рис. 3.2) и можем в них перейти. Но если мы будем это делать без продуманного плана, то можем ходить кругами, а в какую-то часть так и не попасть.

Обойти лабиринт можно с помощью клубка ниток (закрепив конец нитки в исходной точке и нося клубок с собой, мы всегда сможем вернуться) и мела (чтобы отмечать, где мы уже были, и не ходить туда ещё раз). Примерно так же работает алгоритм поиска в глубину. Чтобы хранить пометки, для каждой вершины будем хранить бит, показывающий, были мы в этой вершине или нет. Клубок ниток можно было бы заменить *стеком* (идя в новый коридор,

Рис. 3.2. Лабиринт и его граф.



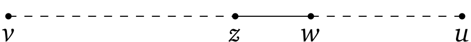
мы добавляем его в стек, а возвращаясь обратно, забираем), но в нашем алгоритме (рис. 3.3) мы вместо явного стека используем рекурсию.[1]

В нашем алгоритме указано также место для процедур Previsit и Postvisit (вызываемых до и после обработки вершины); мы увидим дальше, зачем это может быть полезно.

Рис. 3.3. Нахождение всех вершин, достижимых из данной.

```
процедура Explore(v)
{Вход: вершина v графа G = (V, E).}
{Выход: visited[u] = true для всех вершин u, достижимых из v.}
visited[v] ← true
Previsit(v)
для каждого ребра (v, u) ∈ E:
  если visited[u] = false:   Explore(u)
Postvisit(v)
```

А пока нужно убедиться, что процедура Explore корректна. Ясно, что она обходит только достижимые из v вершины, поскольку на каждом шаге она переходит от вершины к её соседу. Но обходит ли она *все* такие вершины? Допустим, что нашлась достижимая из v вершина u, до которой Explore почему-то не добралась. Рассмотрим тогда какой-нибудь путь из v в u. Пусть z — последняя вершина этого пути, которую посетила процедура Explore (сама вершина v, если других нет), а w — следующая сразу за z вершина на этом пути:



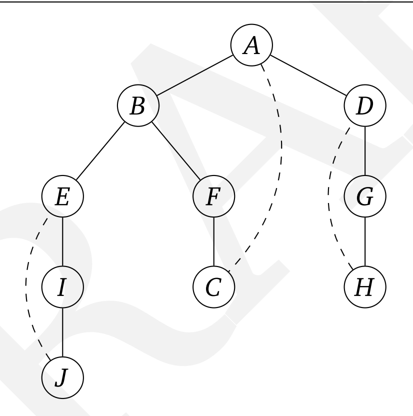
[1]Этот алгоритм (как и многие другие) мы излагаем для ориентированных графов; чтобы использовать его для неориентированных, надо представить неориентированное ребро $\{x, y\}$ парой ориентированных (x, y) и (y, x).

Выходит, что вершину z процедура посетила, а вершину w — нет. Но так быть не может: ведь состоявшийся вызов Explore(z) должен был перебрать всех соседей z.

(Уточнение: начальный вызов процедуры Explore(v) предполагает, что visited[u] = false для всех вершин u; в противном случае она обработает лишь часть графа, где это было так.)

На рис. 3.4 показан вызов Explore для рассмотренного ранее графа и вершины A. Считаем, что соседи вершины перебираются в алфавитном порядке. Сплошными нарисованы рёбра, которые ведут в ранее не встречавшиеся вершины. Например, когда процедура Explore находилась в вершине B, она прошла по ребру B–E, и, поскольку в E она до этого не бывала, был произведён вызов Explore для E. Сплошные рёбра образуют дерево (связный граф без циклов) и поэтому называются *древесными рёбрами* (tree edges). Пунктирные же рёбра ведут в вершины, которые уже встречались. Такие рёбра называются *обратными* (back edges).

Рис. 3.4. Результат вызова Explore(A) для графа с рис. 3.2.



3.2.2. Поиск в глубину

Процедура Explore обходит все вершины, достижимые из данной. Для обхода всего графа алгоритм *поиска в глубину* (рис. 3.5) последовательно вызывает Explore для всех вершин (пропуская уже посещённые).

Рис. 3.5. Поиск в глубину.

```
процедура DFS(G)
для всех вершин v ∈ V:
  visited[v] ← false
для всех вершин v ∈ V:
  если visited[v] = false:   Explore(v)
```

Сразу же отметим, что процедура Explore вызывается для каждой вершины ровно один раз благодаря массиву `visited` (пометки в лабиринте). Сколько времени уходит на обработку вершины? Она включает в себя:

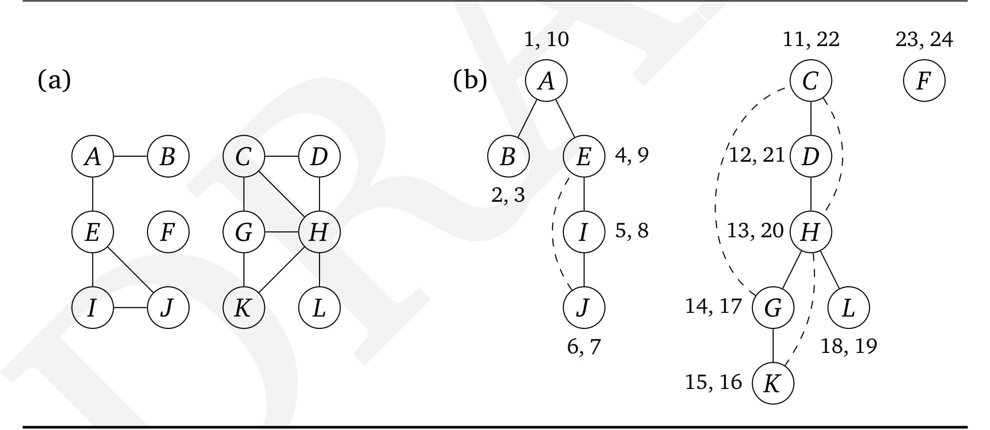
1) $O(1)$-операции: пометка вершины, а также вызовы Previsit и Postvisit (мы не учитываем время внутри этих вызовов);

2) перебор соседей.

Общее время зависит от вершины, но можно оценить количество операций для всех вершин вместе. Общее количество операций на шаге 1 есть $O(|V|)$. На шаге 2 (будем говорить о неориентированном графе) каждое ребро $\{x, y\} \in E$ просматривается ровно *два раза*: при вызовах Explore(x) и Explore(y). Общее время работы на шаге 2, таким образом, есть $O(|E|)$. В целом время работы поиска в глубину есть $O(|V| + |E|)$, то есть линейно. За меньшее время мы не успеем даже прочесть все вершины и рёбра!

На рис. 3.6 показан поиск в глубину на графе с двенадцатью вершинами (не обращайте пока внимания на пары чисел). Опять считаем, что соседи перебираются в алфавитном порядке. Во внешнем цикле процедура Explore вызывается трижды — для вершин A, C и F. Результатом является *лес* (forest) из трёх деревьев с корнями в этих вершинах.

Рис. 3.6. (a) Граф на двенадцати вершинах. (b) Лес, построенный поиском в глубину.



3.2.3. Связные неориентированные графы

Неориентированный граф называется *связным* (connected), если любые две его вершины соединены путём (по рёбрам). Граф на рис. 3.6 не связен: например, нет пути из A в K. Этот граф можно разбить на три непересекающихся связных подмножества:

$$\{A, B, E, I, J\},\ \{C, D, G, H, K, L\},\ \{F\}.$$

Они называются ***компонентами связности*** (connected components). Каждое из них образует связный подграф, а друг с другом они не соединены. Процеду-

ра Explore обходит как раз компоненту связности той вершины, для которой она была вызвана. При каждом вызове Explore во внешнем цикле алгоритма DFS обходится новая компонента связности. Таким образом, с помощью поиска в глубину легко проверить связность графа. Более того, можно для каждой вершины v найти номер её компоненты связности $\texttt{ccnum}[v]$ (в порядке обнаружения). Для этого нужно завести переменную `cc`, изначально равную нулю, и увеличивать её на единицу каждый раз, когда Explore вызывается во внешнем цикле, а процедуру Previsit сделать такой:

процедура Previsit(v)
$\texttt{ccnum}[v] \leftarrow \texttt{cc}$

3.2.4. Время начала и конца обработки вершины

Как мы видим, поиск в глубину позволяет за линейное время определить, связен ли граф. И это далеко не всё — сейчас мы рассмотрим приём, который лежит в основе многих других применений поиска в глубину. Будем записывать время начала обработки каждой вершины (вызов Previsit) и время конца обработки (Postvisit). Для нашего примера эти числа (для всех 24 таких событий) показаны на рис. 3.6. Например, в момент времени 5 началась обработка вершины I, а в момент времени 21 закончилась обработка вершины D.

Чтобы сохранить эту информацию при поиске в глубину, заведём счётчик `clock`, изначально равный 1, и напишем так:

процедура Previsit(v)
$\texttt{pre}[v] \leftarrow \texttt{clock}$
$\texttt{clock} \leftarrow \texttt{clock} + 1$

процедура Postvisit(v)
$\texttt{post}[v] \leftarrow \texttt{clock}$
$\texttt{clock} \leftarrow \texttt{clock} + 1$

Вот простое (но важное) свойство этих чисел:

Свойство. *Для любых двух вершин u и v либо отрезки $[\texttt{pre}(u), \texttt{post}(u)]$ и $[\texttt{pre}(v), \texttt{post}(v)]$ не пересекаются, либо один содержится в другом.*

В самом деле, $[\texttt{pre}(u), \texttt{post}(u)]$ — это промежуток времени, в течение которого вершина u была в стеке, и если вершина v была помещена в стек, когда там уже лежала вершина u, то v будет вынута раньше u.

Таким образом, рассмотренные отрезки отражают структуру дерева, построенного при поиске в глубину. Это особенно полезно для ориентированных графов, к которым мы и переходим.

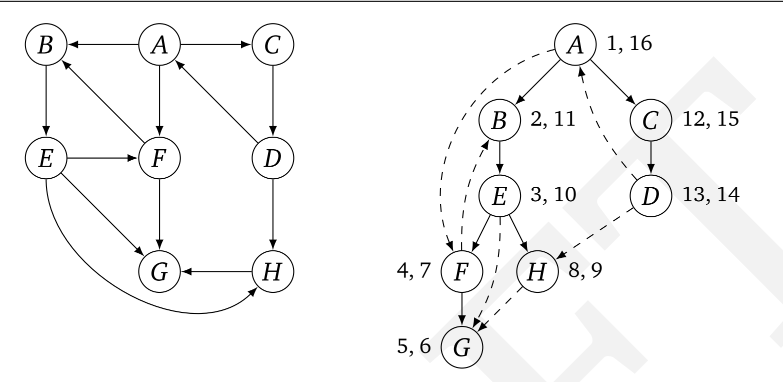
3.3. Поиск в глубину в ориентированных графах

3.3.1. Типы рёбер

Рассмотренный нами алгоритм поиска в глубину может быть использован и для ориентированных графов (в процедуре Explore надо перебирать выходя-

щие из вершины рёбра). На рис. 3.7 показан пример ориентированного графа и дерева, построенного поиском в глубину (напомним, что соседи перебираются в алфавитном порядке).

Рис. 3.7. Поиск в глубину для ориентированных графов.



Напомним общеизвестную терминологию, связанную с деревьями. Вершина A называется *корнем дерева* (tree root), все остальные вершины этого дерева являются её *потомками* (descendants). Вершины F, G, H являются потомками E, а E является их *предком* (ancestor). Наконец, C является *родителем* (parent) D, а D — *ребёнком* (child) C. (В отличие от людей, у вершины дерева может быть только один родитель.)

При поиске в глубину в неориентированных графах мы различали древесные рёбра дерева и обратные (все остальные). Для ориентированных графов типов рёбер будет уже четыре.

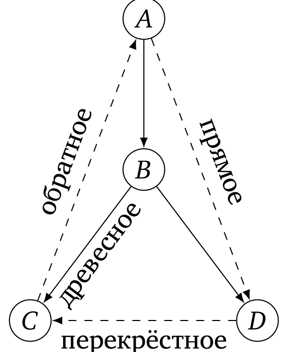
Древесные рёбра (tree edges) — рёбра леса, построенного поиском в глубину.

Прямые рёбра (forward edges) ведут от вершины к её потомку, не являющемуся при этом её ребёнком.

Обратные рёбра (back edges) ведут от вершины к её предку.

Перекрёстные рёбра (cross edges) ведут от вершины к другой вершине, не являющейся ни предком, ни потомком первой (такая вершина уже полностью обработана в момент обнаружения перекрёстного ребра).





На рис. 3.7 можно обнаружить два прямых, два обратных и два перекрёстных ребра (найдите их).

Тип ребра можно узнать по pre- и post-значениям его концов. Легко видеть, что вершина u является предком v тогда и только тогда, когда v была обнаружена во время вызова EXPLORE(u). В этом случае $\texttt{pre}[u] < \texttt{pre}[v] < \texttt{post}[v] < \texttt{post}[u]$, то есть отрезок обработки v вложен в отрезок обработки u. Для обратных рёбер: u — потомок v тогда и только тогда, когда v — предок u; для перекрёстных рёбер u–v отрезки не пересекаются (и v-отрезок предшествует u-отрезку):

pre/post-отрезки для (u, v)	*типы рёбер*
$[_u \; [_v \;]_v \;]_u$	древесное/прямое
$[_v \; [_u \;]_u \;]_v$	обратное
$[_v \;]_v \; [_u \;]_u$	перекрёстное

Понятно ли, почему не бывает других случаев расположения отрезков?

3.3.2. Ориентированные ациклические графы

Циклом (cycle) в ориентированном графе называется путь $v_0 \to v_1 \to v_2 \to \ldots \ldots \to v_k \to v_0$ из некоторой вершины в неё же (по рёбрам). Скажем, в графе рис. 3.7 есть цикл $B \to E \to F \to B$ и несколько других. Граф без циклов называется *ациклическим* (acyclic). Наличие циклов в ориентированном графе легко проверить с помощью одного вызова поиска в глубину.

Свойство. *Ориентированный граф содержит цикл тогда и только тогда, когда при поиске в глубину обнаруживается обратное ребро.*

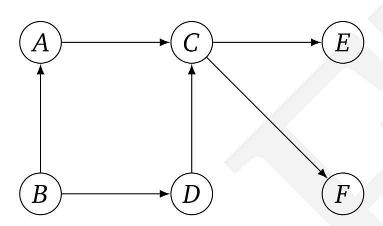
Доказательство. Ясно, что если (u, v) — обратное ребро, то путь в дереве поиска в глубину от v к u и само ребро (u, v) образуют цикл.

Обратно: пусть граф содержит цикл $v_0 \to v_1 \to \ldots \to v_k \to v_0$ и пусть v_i — вершина этого цикла, которая была обнаружена *первой* (то есть вершина цикла с минимальным $\texttt{pre}[v_i]$). Все оставшиеся вершины цикла достижимы из v_i, и поэтому в дереве поиска в глубину они будут потомками v_i. Тогда ребро $v_{i-1} \to v_i$ (или же $v_k \to v_0$, если $i = 0$) ведёт из вершины в её предка, то есть является обратным ребром. □

С помощью *ориентированных ациклических графов* (по-английски их называют directed acyclic graphs или сокращённо dags) удобно представлять временные и иерархические отношения, а также причинно-следственные связи. Допустим, к примеру, что необходимо сделать несколько вещей, причём есть ограничения на порядок (сначала надо проснуться и лишь потом встать с кровати; душ надо принять после того, как встал с кровати, но до того, как оделся). Как выбрать правильный порядок дел?

Ограничения представим как ориентированный граф: каждому делу соответствует вершина; ребро из u в v означает, что u должно быть выполнено до v. Если в этом графе есть цикл, то правильного порядка не существует вовсе. Если же граф ациклический, то есть надежда его *линеаризовать* (linearize) или, как ещё говорят, *топологически упорядочить* (topologically sort) — так упорядочить вершины, чтобы каждое ребро вело от более ранней вершины к более поздней. Например, для графа рис. 3.8 один из допустимых порядков таков: B, A, D, C, E, F. (Видите ли вы ещё три порядка?)

Рис. 3.8. Ориентированный граф с одним источником, двумя стоками и четырьмя возможными линеаризациями.



Какие же ориентированные ациклические графы можно топологически упорядочить? Оказывается, что *все*, и это можно сделать с помощью поиска в глубину, расположив вершины в порядке *убывания* их post-значений. Ведь $\texttt{post}(u) < \texttt{post}(v)$ только для обратных рёбер (u, v) (см. таблицу на с. 91), а мы уже знаем, что в ациклическом графе обратных рёбер быть не может:

Свойство. *В ориентированном ациклическом графе каждое ребро идёт из вершины с бо́льшим* post*-значением в вершину с меньшим* post*-значением.*

Посмотрим на первую и последнюю вершину после топологической сортировки. Из последней вершины рёбра не выходят: как говорят, она является *стоком* (sink). Напротив, в первую вершину рёбра не входят: как говорят, она является *истоком* (source). В частности, мы доказали такое утверждение:

Свойство. *У каждого ациклического ориентированного графа есть хотя бы один исток и хотя бы один сток.*

Это свойство (которое несложно доказать и без поиска в глубину) можно использовать для линеаризации: пока в графе есть вершины, надо искать исток, печатать его и удалять из графа. Понятно ли вам, почему такой алгоритм работает? Что будет, если в графе есть цикл? Как реализовать такой алгоритм за линейное время (упражнение 3.14)?

3.4. Компоненты сильной связности

3.4.1. Связность для ориентированных графов

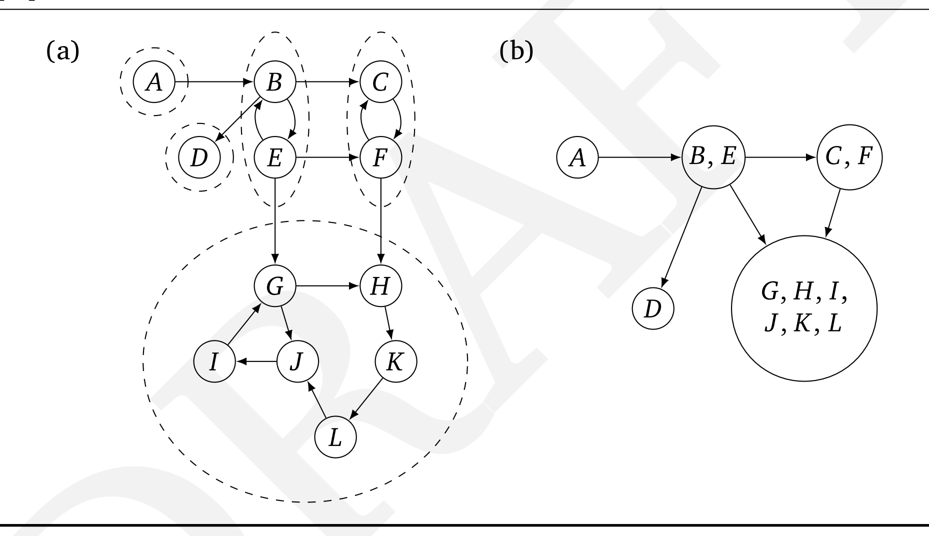
Что означает связность для ориентированного графа? Тут надо быть аккуратным. Несвязный *неориентированный* граф можно разбить на связные компо-

ненты, не соединённые друг с другом. С *ориентированными* графами ситуация сложнее. Граф рис. 3.9 нельзя разбить на две части, не соединённые друг с другом. Но мы не будем называть этот граф связным, поскольку в нём нет пути из G в B или из F в A. Дадим такое определение:

> Вершины u и v ориентированного графа называются *связанными* (connected), если в нём есть путь из u в v, *а также* путь из v в u.

Такое отношение на вершинах разбивает все множество вершин на непересекающиеся подмножества (упражнение 3.30), называемые *компонентами сильной связности* (strongly connected components). Граф рис. 3.9 имеет пять таких компонент.

Рис. 3.9. (a) Ориентированный граф и его компоненты сильной связности. (b) Метаграф.



Стянем теперь каждую компоненту сильной связности в отдельную вершину (*метавершину*) и оставим только рёбра между метавершинами (см. рис. 3.9), удалив дубликаты. Полученный граф называется *метаграфом* (meta-graph) исходного.[1] Он не содержит циклов: если бы несколько компонент образовали цикл, то вершины этих компонент были бы в исходном графе достижимы друг из друга и вошли бы в одну компоненту.

Свойство. *Метаграф любого ориентированного графа является ациклическим (и может быть топологически упорядочен).*

Но как построить метаграф для данного графа?

[1]В русской литературе такой граф называется также *графом компонент* или *конденсацией* исходного графа. — *Прим. перев.*

3.4.2. Алгоритм построения метаграфа

Компоненты сильной связности и метаграф могут быть построены за линейное время. Сейчас мы увидим, как это делается (и нам пригодится поиск в глубину). Начнём с такого замечания (уже нам известного).

Свойство 1. *Процедура* Explore, *вызванная для вершины u, заканчивает работу, когда посещены все вершины, достижимые из u.*

Значит, если вызвать Explore для вершины, которая лежит в *компоненте-стоке* метаграфа, то мы обойдём как раз все вершины этой компоненты. Например, граф рис. 3.9 имеет две такие компоненты, и вызов Explore для вершины K обойдёт бо́льшую из них.

Остаётся понять, (а) как найти вершину, которая гарантированно лежит в компоненте-стоке, и (б) что делать после нахождения компоненты-стока.

Для начала ответим на первый вопрос. Сначала покажем, как решать симметричную задачу: найти вершину в *компоненте-истоке*.

Свойство 2. *Вершина, которой поиск в глубину присваивает максимальное* post*-значение, лежит в компоненте-истоке.*

Это вытекает из следующего более общего факта.

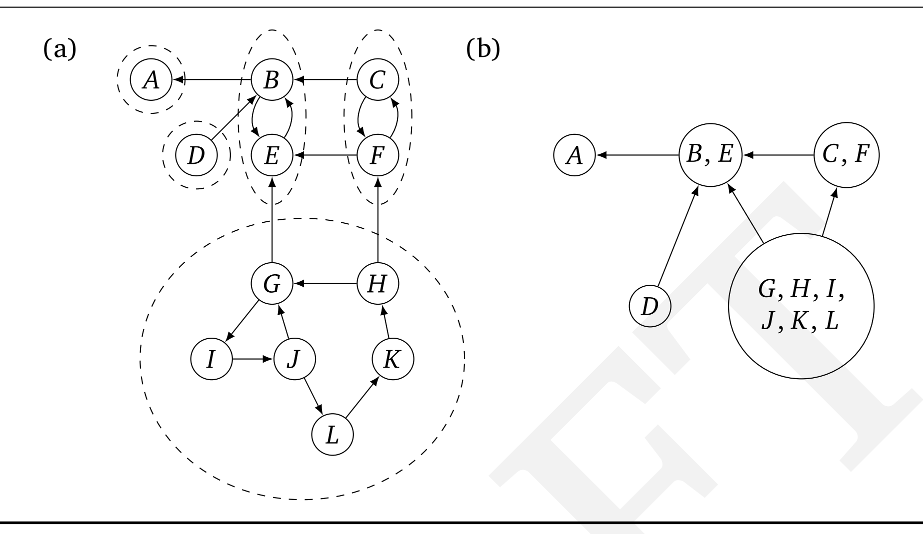
Свойство 3. *Пусть C и C' — компоненты сильной связности графа и в графе есть ориентированное ребро из C в C'. Тогда максимальное* post*-значение вершин в C больше, чем максимальное* post*-значение вершин в C'.*

Доказательство. Дождёмся момента, когда при поиске в глубину впервые появится вершина v из C или C'. Если $v \in C$, то вызов Explore(v) не завершится, пока не будут обработаны все вершины обеих компонент (см. свойство 1). Поэтому $\texttt{post}[v]$ будет больше, чем у всех вершин из C'. Если же $v \in C'$, то вызов Explore(v) обойдёт все вершины C', но до C дело ещё не дойдёт — и максимальное post-значение в C тоже будет больше. □

Свойство 3 можно переформулировать так: *компоненты сильной связности можно топологически упорядочить, расположив их по убыванию максимальных* post*-значений вершин в них.* Это наблюдение обобщает рассмотренный нами алгоритм топологической сортировки ориентированных ациклических графов (в таких графах каждая компонента сильной связности состоит из одной вершины).

Свойство 2 позволяет найти нам вершину в компоненте-истоке графа, но нам хотелось бы иметь вершину в *компоненте-стоке*. Поэтому мы рассмотрим *обращённый* (reverse) граф G^R, получаемый из G изменением направлений всех рёбер (см. рис. 3.10). У G^R будут те же компоненты сильной связности, как и у G (почему?). Поиск в глубину на G^R даст максимальное post-значение для вершины из компоненты-истока графа G^R, то есть для вершины из компоненты-стока графа G. Мы, таким образом, ответили на вопрос (а).

Рис. 3.10. Обращённый граф рис. 3.9.



Перейдём к вопросу (б). Как же находить следующую компоненту, когда компонента-сток уже найдена? В этом нам опять поможет свойство 3. Когда компонента вершины с максимальным `post`-значением найдена и удалена из графа, вершина с максимальным `post`-значением среди оставшихся вершин (имеются в виду уже вычисленные значения, мы не запускаем поиск повторно!) будет опять принадлежать компоненте-стоку в оставшемся графе, и так далее. Итоговый алгоритм прост:

1. Вызвать поиск в глубину для G^R.

2. Вызвать алгоритм нахождения компонент связности в неориентированных графах (см. п. 3.2.3), перебирая в поиске в глубину вершины в порядке убывания их `post`-значений, найденных на первом шаге.

Время работы приведённого алгоритма линейно; он примерно вдвое медленнее поиска в глубину. (Разберитесь, как строить списки смежности графа G^R и как упорядочить вершины по убыванию `post`-значений за линейное время.)

Опробуем этот алгоритм на графе рис. 3.9. Если на первом шаге вершины перебираются в алфавитном порядке, то убывание `post`-значений при поиске в глубину на G^R идёт так: G, I, J, L, K, H, D, C, F, B, E, A. Второй шаг обнаруживает компоненты в таком порядке: $\{G, H, I, J, K, L\}$, $\{D\}$, $\{C, F\}$, $\{B, E\}$, $\{A\}$.

Быстрый обход веб-страниц

В этой главе мы предполагали, что граф дан нам в удобном виде: вершины пронумерованы от 1 до n, рёбра хранятся в списках смежности. Для веб-

графа всё гораздо сложнее: вершины заранее не известны, они обнаруживаются одна за другой в процессе поиска. И рекурсию использовать, конечно же, нельзя (глубина рекурсии будет слишком большой). Тем не менее для обхода веб-страниц используют алгоритмы, очень похожие на поиск в глубину. Поддерживается стек, содержащий все вершины, которые обнаружены (упоминаются в просмотренных ссылках), но пока не обработаны. В действительности это не совсем стек, поскольку первыми из него достаются не те вершины, которые были помещены позже всех (и даже не вершины, которые попали туда раньше всех, как при *поиске в ширину*, см. главу 4). Первыми обрабатываются вершины, которые выглядят наиболее «интересными» (по каким-то эвристическим критериям). Это нужно, чтобы стек не переполнялся и чтобы в худшем случае необработанными остались только вершины, которые вряд ли бы привели к существенному расширению просмотренной области.

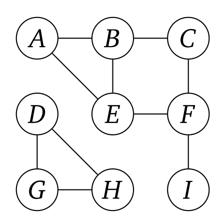
Обход веб-страниц обычно осуществляется несколькими компьютерами, вызывающими EXPLORE параллельно: каждый из них снимает вершину со стека, скачивает соответствующую страницу и ищет в ней гиперссылки. Когда по ссылке находится новая страница, вместо рекурсивного вызова новая вершина просто помещается в стек (общий для всех компьютеров).

Один момент по-прежнему остаётся непонятным: когда находится новая страница, откуда мы знаем, что она действительно новая и что мы ещё не обрабатывали её? И какое *имя* мы даём ей (добавляем в стек и запоминаем как обработанное)? Это делается с помощью *хеширования*.

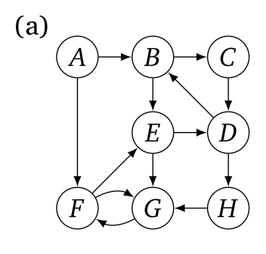
Кстати, оказалось, что алгоритмы нахождения сильно связанных компонент выявляют интересные свойства веб-графа.

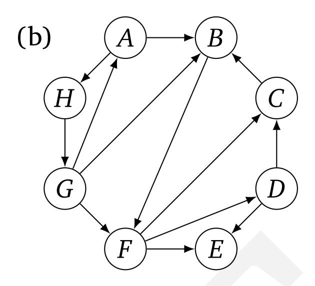
Упражнения

3.1. Примените алгоритм поиска в глубину к данному графу, перебирая вершины в алфавитном порядке. Пометьте каждое ребро как древесное или обратное, а также вычислите `pre`- и `post`-значения для каждой вершины.

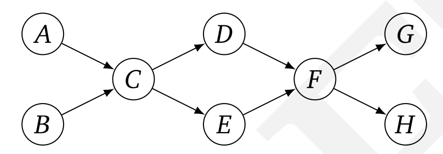


3.2. Примените алгоритм поиска в глубину к двум приведённым ниже графам, перебирая вершины в алфавитном порядке. Пометьте каждое ребро как древесное, прямое, обратное или перекрёстное, а также вычислите `pre`- и `post`-значения для каждой вершины.





3.3. Примените алгоритм топологической сортировки, основанный на поиске в глубину, к следующему графу. Каждый раз в качестве следующей выбирайте первую в алфавитном порядке вершину.



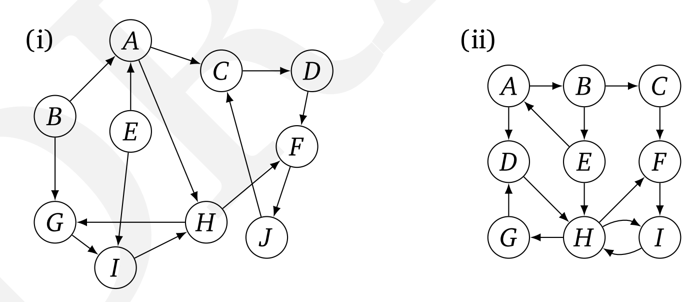
(a) Найдите `pre`- и `post`-значения всех вершин.

(b) Найдите все истоки и стоки графа.

(c) Какая линеаризация графа у вас получилась?

(d) Сколько различных линеаризаций есть у этого графа?

3.4. Примените алгоритм нахождения компонент сильной связности к следующим графам G. При поиске в глубину на G^R в качестве следующей выбирайте первую в алфавитном порядке вершину.



(a) В каком порядке алгоритм находит компоненты сильной связности?

(b) Какие компоненты являются стоками и какие — истоками?

(c) Нарисуйте метаграф (его вершинами являются компоненты сильной связности G).

(d) Какое минимальное число рёбер надо добавить, чтобы сделать граф сильно связным?

3.5. *Обращённым* графом для графа $G = (V, E)$ называется граф $G^R = (V, E^R)$ с теми же вершинами, рёбра которого получены изменением направления всех рёбер исходного графа: $E^R = \{(v, u) : (u, v) \in E\}$.

Приведите линейный по времени алгоритм, который по заданному в виде списка смежности графу строит обращённый граф (тоже в виде списка смежности).

3.6. *Степенью* (degree) $d(v)$ вершины v неориентированного графа G называется количество соседей v или, как говорят, *инцидентных* v рёбер. Для ориентированных графов различают *входящую степень* (indegree) $d_{\text{in}}(v)$ и *исходящую степень* (outdegree) $d_{\text{out}}(v)$, то есть количество входящих в v и исходящих из v рёбер (соответственно).

(a) Докажите, что для неориентированного графа выполнено равенство $\sum\limits_{u \in V} d(u) = 2|E|$.

(b) Выведите отсюда, что в неориентированном графе количество вершин нечётной степени чётно.

(c) Верно ли это утверждение для вершин нечётной входящей степени в ориентированном графе?

3.7. Граф $G = (V, E)$ называется *двудольным* (bipartite), если множество его вершин можно разделить на две части ($V = V_1 \cup V_2$ и $V_1 \cap V_2 = \varnothing$), так что все рёбра соединяют вершины из разных частей (например, если $u, v \in V_1$, то u и v не соединены ребром).

(a) Постройте линейный по времени алгоритм, определяющий, является ли входной граф двудольным.

(b) Определение двудольного графа можно давать разными способами. Например, граф является двудольным тогда и только тогда, когда его вершины можно корректно раскрасить в два цвета так, чтобы каждое ребро соединяло вершины разного цвета.

Докажите, что неориентированный граф является двудольным тогда и только тогда, когда в нём нет циклов нечётной длины.

(c) Сколько цветов понадобится, чтобы покрасить любой неориентированный граф, содержащий *ровно один* цикл нечётной длины?

3.8. *Переливание воды.* Есть три сосуда ёмкостью 10, 7 и 4 литров. Изначально первый сосуд пуст, а оставшиеся два полностью наполнены водой. За один ход разрешается переливать из одного сосуда воду в другой до тех пор, пока первый не станет пустым или же второй не заполнится до верха. Мы хотим проверить, существует ли последовательность ходов, в результате которой в одном из последних двух сосудов останется ровно два литра воды.

(a) Сформулируйте данную задачу как задачу о графах: определите соответствующий граф и переформулируйте вопрос в его терминах.

(b) Какой алгоритм нужно применить, чтобы ответить на данный вопрос?

(c) Найдите ответ, применив алгоритм.

3.9. Для вершины u неориентированного графа зададим $\texttt{twodegree}[u]$ как сумму степеней соседей u. Покажите, как заполнить массив $\texttt{twodegree}[\cdot]$ за линейное время для графа, заданного списком смежности.

3.10. Перепишите процедуру Explore (рис. 3.3) без использования рекурсии. Процедуры Previsit и Postvisit должны вызываться в том же порядке, что в рекурсивной процедуре.

3.11. Укажите линейный алгоритм, который по данному ориентированному графу G и его ребру e определяет, есть ли в G цикл, содержащий e.

3.12. Докажите или опровергните: если $\{u, v\}$ — ребро неориентированного графа и $\texttt{post}(u) < \texttt{post}(v)$, то v является предком u в дереве поиска в глубину.

3.13. *Связность неориентированных и ориентированных графов.*

(a) Докажите, что в каждом связном неориентированном графе найдётся вершина, удаление которой оставляет граф связным. (Подсказка: рассмотрите дерево поиска в глубину графа.)

(b) Приведите пример сильно связного ориентированного графа, который перестаёт быть сильно связным при удалении любой вершины.

(c) Неориентированный граф, состоящий из двух компонент связности, всегда можно сделать связным, добавив одно ребро. Приведите пример ориентированного графа, состоящего из двух компонент сильной связности, из которого нельзя получить сильно связный граф добавлением одного ребра.

3.14. На с. 92 обсуждался альтернативный алгоритм топологической сортировки графа, который последовательно удаляет из графа истоки. Приведите линейную по времени реализацию такого алгоритма. (Подсказка: можно хранить входящие степени всех вершин и множество вершин нулевой входящей степени, корректируя эти данные по мере удаления вершин нулевой степени.)

3.15. (a) Власти сделали все дороги города односторонними и утверждают, что от любого перекрёстка по-прежнему можно добраться до любого другого, не нарушая правила. Как быстро их разоблачить? Сформулируйте данную задачу на языке графов и покажите, как её можно решить за линейное время.

(b) В ответ на претензии власти заявили, что имели в виду другое: куда бы ни поехать от мэрии по правилам, можно будет вернуться, не нарушая правил. Как проверить это утверждение за линейное время?

3.16. Магистерская программа по информатике состоит из n семестровых курсов. Граф G отражает зависимости: вершины графа соответствуют курсам, из v идёт ориентированное ребро в w, если w можно изучать только после v. Постройте линейный по времени алгоритм, который по G определяет минимальное количество семестров, необходимое для изучения всей программы (в одном семестре может быть сколько угодно курсов).

3.17. *Бесконечные пути.* Пусть дан ориентированный граф $G = (V, E)$ с выделенной «начальной» вершиной $s \in V$, а также множествами «хороших» вершин $V_G \subseteq V$ и «плохих» вершин $V_B \subseteq V$. *Бесконечным путём* (infinite trace) в графе G называется бесконечная последовательность $v_0 v_1 v_2 \ldots$ вершин, в ко-

торой $v_0 = s$ и $(v_i, v_{i+1}) \in E$ для всех $i \geqslant 0$. В некоторых вершинах такой путь побывает бесконечно много раз.

(a) Через $\mathrm{Inf}(p) \subseteq V$ обозначим те вершины бесконечного пути p, по которым путь проходит бесконечное число раз. Покажите, что $\mathrm{Inf}(p)$ целиком попадает в одну из компонент сильной связности графа G.

(b) Постройте алгоритм, определяющий, есть ли в G бесконечный путь.

(c) Постройте алгоритм, определяющий, есть ли в G бесконечный путь, проходящий через хорошие вершины бесконечно много раз.

(d) Наконец, постройте алгоритм, проверяющий, существует ли бесконечный путь, который проходит бесконечное количество раз хотя бы по одной хорошей вершине, но по всем плохим вершинам проходит лишь конечное число раз.

3.18. Пусть двоичное дерево $T = (V, E)$ задано списком смежности и указан корень дерева $r \in V$. Напомним, что вершина дерева u называется *предком* вершины v, если путь от r до v в T проходит через u.

Мы хотим выполнить линейную предобработку дерева так, чтобы после этого была возможность отвечать на вопросы типа «является ли u предком v?» за время $O(1)$. Возможно ли это?

3.19. Пусть опять дано дерево $T = (V, E)$ с выделенным корнем, а также массив $x[\cdot]$, содержащий по числу для каждой вершины дерева. Определим новый массив $z[\cdot]$:

$$z[u] = \text{максимальное значение массива } x \text{ на потомках } u.$$

Покажите, как заполнить такой массив за линейное время.

3.20. Дано дерево $T = (V, E)$ с выделенным корнем $r \in V$. *Родителем* вершины дерева v называется предпоследняя вершина на пути из r в v. Считаем также, что $p(r) = r$. Пусть $p^1(v) = p(v)$ и $p^k(v) = p^{k-1}(p(v))$ при $k > 1$. Другими словами, $p^k(v)$ — это предок v в k-м поколении. Пусть также каждая вершина дерева v помечена положительным целым числом $l[v]$. Приведите линейный алгоритм, находящий новые метки $l_{\text{new}}(v) = l(p^{l(v)}(v))$.

3.21. Постройте линейный алгоритм нахождения цикла нечётной длины в *ориентированном* графе. (Подсказка: рассмотрите сначала случай сильно связного графа.)

3.22. Постройте эффективный алгоритм, который определяет, есть ли в данном ориентированном графе вершина, из которой достижимы все вершины графа.

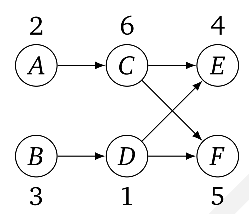
3.23. Постройте эффективный алгоритм, который по двум вершинам $s, t \in V$ ориентированного ациклического графа $G = (V, E)$ находит количество различных путей из s в t.

3.24. Постройте линейный алгоритм, который проверяет, есть ли в данном ориентированном ациклическом графе путь, проходящий через каждую вершину ровно один раз.

3.25. Дан ориентированный граф, каждой вершине $u \in V$ которого приписана *цена* (price) p_u, представляющая собой целое положительное число. Определим массив cost следующим образом:

$$\texttt{cost}[u] = \text{минимальная цена вершин, достижимых из } u \text{ (включая } u\text{).}$$

Например, значения массива cost для вершин A, B, C, D, E, F графа на рисунке равны 2, 1, 4, 1, 4, 5. (Цены p_u указаны около вершин.)



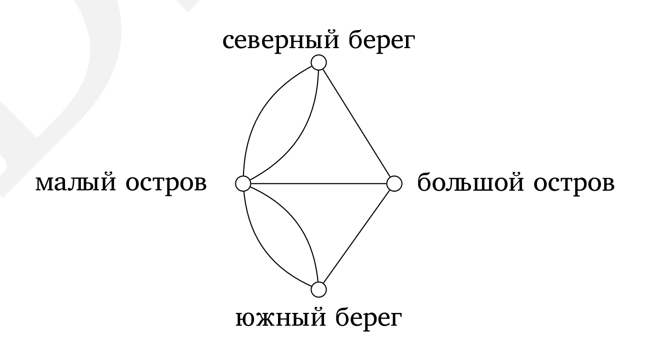
Необходимо заполнить массив cost для всех вершин графа.

(a) Постройте линейный алгоритм для ориентированных *ациклических* графов. (Подсказка: обрабатывайте вершины в определённом порядке.)

(b) Обобщите алгоритм так, чтобы он (по-прежнему за линейное время) решал задачу для произвольных ориентированных графов. (Подсказка: вспомните о метаграфе из компонент сильной связности.)

3.26. *Эйлеровым циклом* (Eulerian tour) неориентированного графа называется замкнутый путь, проходящий по всем рёбрам графа ровно по одному разу. В 1736 году Леонард Эйлер решил знаменитую задачу о кёнигсбергских мостах, положившую начало теории графов. В Кёнигсберге (в настоящее время Калининград, Россия) два острова и два берега реки были связаны семью мостами, и возник вопрос: можно ли обойти все эти семь мостов и вернуться в исходную точку, пройдя по каждому из мостов *ровно один раз*?

Соответствующий граф приведён ниже. Отметим, что в нём есть кратные рёбра (несколько рёбер между парой вершин; мы такого не допускали).



(a) Докажите, что в неориентированном графе есть эйлеров цикл тогда и только тогда, когда он связен и степени всех его вершин чётны.

(b) *Эйлеровым путём* (Eulerian path) называется путь в графе, проходящий по каждому ребру графа ровно один раз. Для каких графов такой путь существует?

(c) При каких условиях ориентированный граф имеет эйлеров цикл?

3.27. Два пути в графе называются *не пересекающимися по рёбрам* (edge-disjoint), если они не содержат общих рёбер. Покажите, что в любом неориентированном графе вершины нечётной степени можно объединить в пары и соединить вершины каждой пары путём таким образом, чтобы все эти пути не пересекались по рёбрам.

3.28. Входом *задачи 2-выполнимости* (2-satisfiability problem, 2SAT) является конъюнкция *дизъюнктов* (clauses), каждый из которых является дизъюнкцией (логическое ИЛИ) двух *литералов*, где литерал — это булева переменная или её отрицание. Требуется выяснить, можно ли всем переменным формулы присвоить значения `true` и `false`, выполнив *все* дизъюнкты. Дизъюнкт является выполненным, если хотя бы один его литерал имеет значение `true`. Например, чтобы выполнить формулу

$$(x_1 \lor \bar{x}_2) \land (\bar{x}_1 \lor \bar{x}_3) \land (x_1 \lor x_2) \land (\bar{x}_3 \lor x_4) \land (\bar{x}_1 \lor x_4),$$

можно присвоить переменным x_1, x_2, x_3, x_4 значения `true`, `false`, `false`, `true` (соответственно).

(a) Найдите все выполняющие наборы этой формулы.

(b) Придумайте пример задачи 2-выполнимости с четырьмя переменными, для которой нет выполняющего набора.

Сейчас мы построим эффективный алгоритм проверки 2-выполнимости, основанный на поиске компонент сильной связности. По входу I задачи 2-выполнимости с n переменными и m дизъюнктами построим ориентированный граф $G_I = (V, E)$ следующим образом.

- Заведём по вершине для каждой переменной и для отрицания каждой переменной. Всего, таким образом, будет $2n$ вершин.
- Для каждого дизъюнкта $(\alpha \lor \beta)$ формулы I проведём ребро из отрицания α в β, а также ребро из отрицания β в α (через α и β здесь обозначены литералы, отрицанием отрицания переменной считается сама переменная). Всего получится $2m$ рёбер.

Отметим, что дизъюнкт $(\alpha \lor \beta)$ эквивалентен как импликации $\bar{\alpha} \Rightarrow \beta$, так и импликации $\bar{\beta} \Rightarrow \alpha$. То есть в G_I мы просто записали импликации, следующие из I.

(c) Постройте граф для формулы из примера выше, а также для формулы, построенной вами в пункте (b).

(d) Покажите, что если в G_I есть компонента сильной связности, содержащая одновременно x и $\bar{x}$ для некоторой переменной x, то I невыполнима.

(e) Покажите теперь обратное: если ни одна компонента сильной связности графа G_I не содержит одновременно литерала и его отрицания, то фор-

мула выполнима. (Подсказка: покажите, что следующий алгоритм строит выполняющий набор для формулы. Выбираем в метаграфе G_I компоненту-сток и присваиваем значение `true` всем литералам из неё и значение `false` отрицаниям всех таких литералов, после чего выкидываем все литералы, у которых уже есть значения.)

(f) Наконец, используя все приведённые выше идеи, покажите, что задача 2-выполнимости решается за линейное время.

3.29. Бинарным *отношением* (binary relation) на конечном множестве S называется множество R упорядоченных пар $(x, y) \in S \times S$. Если, например, S является множеством людей, то пара $(x, y) \in R$ может означать, что x знает y. *Отношением эквивалентности* (equivalence relation) называется бинарное отношение, удовлетворяющее трём свойствам:

- рефлексивность (reflexivity): $(x, x) \in R$ для всех $x \in S$;
- симметричность (symmetry): если $(x, y) \in R$, то и $(y, x) \in R$;
- транзитивность (transitivity): если $(x, y) \in R$ и $(y, z) \in R$, то и $(x, z) \in R$.

Например, отношение «у x и y дни рождения в один день» является отношением эквивалентности, а «x является отцом y» — нет (не выполняется ни одно из трёх условий).

Покажите, что отношение эквивалентности разбивает множество S на непересекающиеся множества $S_1, S_2, \ldots, S_k$ (то есть $S = S_1 \cup S_2 \cup \ldots \cup S_k$ и $S_i \cap S_j = \varnothing$ для всех $i \neq j$), при этом

- любые два элемента одного подмножества эквивалентны: $(x, y) \in R$ для любых x, y из любого S_i;
- элементы же разных подмножеств не эквивалентны: $(x, y) \notin R$ для любых $x \in S_i$ и $y \in S_j$ при $i \neq j$.

(Подсказка: представьте отношение эквивалентности как неориентированный граф.)

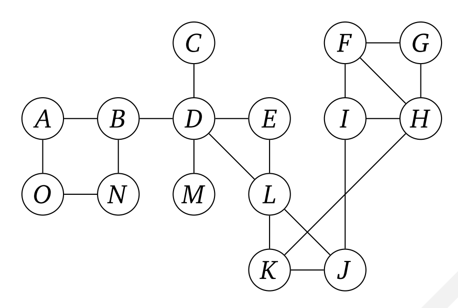
3.30. На с. 93 мы ввели отношение «x связан с y» на множестве вершин *ориентированного* графа. Покажите, что оно является отношением эквивалентности (см. предыдущее упражнение) и разбивает граф на компоненты сильной связности.

3.31. *Компоненты двусвязности.* Рассмотрим неориентированный граф $G = (V, E)$. Будем говорить, что $e \sim e'$ для двух рёбер $e, e' \in E$, если $e = e'$ или же существует простой цикл, содержащий одновременно e и e'.

(a) Покажите, что отношение $\sim$ является отношением эквивалентности (см. упражнение 3.29).

Классы эквивалентности, на которые данное отношение разбивает множество рёбер, называются *компонентами двусвязности* (biconnected components) графа G. *Мостом* (bridge) называется ребро, в компоненту двусвязности которого входит только оно само. *Точкой сочленения* (separating vertex) называется вершина, удаление которой делает граф несвязным.

(b) Разбейте рёбра приведённого ниже графа на компоненты двусвязности, найдите мосты и точки сочленения.



Таким образом, мы получаем разбиение рёбер на классы эквивалентности (двусвязные компоненты); одновременно мы *почти что* разбиваем вершины графа на классы. А именно:

(c) Для каждой компоненты двусвязности рассмотрим множество концов входящих в эту компоненту рёбер. Покажите, что эти множества для двух различных компонент двусвязности либо не пересекаются вовсе, либо пересекаются по точке сочленения.

(d) Стянем теперь каждую компоненту двусвязности в отдельную «метавершину», сохранив также все точки сочленения (таким образом, метавершина-компонента будет соединена со всеми её точками сочленения). Покажите, что полученный граф является деревом.

Компоненты двусвязности, мосты и точки сочленения графа можно найти за линейное время при помощи поиска в глубину.

(e) Покажите, что корень дерева поиска в глубину является точкой сочленения тогда и только тогда, когда у него более одного ребёнка.

(f) Покажите, что некорневая вершина v дерева поиска в глубину является точкой сочленения тогда и только тогда, когда у неё есть ребёнок v', для которого ни v', ни его потомки не имеют обратного ребра, ведущего в предка вершины v (само v не считается своим предком).

(g) Для каждой вершины u положим $\texttt{low}(u)$ равным минимальному значению среди $\texttt{pre}(u)$ и всех значений $\texttt{pre}(w)$ для всех обратных рёбер (v, w), у которых v — потомок u. Покажите, что массив значений low можно заполнить за линейное время.

(h) Покажите, как найти компоненты двусвязности, мосты и точки сочленения графа за линейное время. (Подсказка: используйте значения массива low для нахождения точек сочленения. Запустите поиск в глубину с дополнительным стеком рёбер, чтобы по одной удалять компоненты двусвязности.)

Глава 4

Пути в графах

4.1. Расстояния в графе

Поиск в глубину не только быстро находит все вершины, достижимые из начальной, но и строит дерево, содержащее пути к ним (рис. 4.1). Однако эти пути не обязательно будут *кратчайшими*: например, от S до C можно добраться по одному ребру, в то время как поиск в глубину находит путь из трёх. А как искать кратчайшие, мы изучим в этой главе.

Расстоянием (distance) между двумя вершинами неориентированного графа будем называть длину кратчайшего пути между ними, измеренную в рёбрах. У этого понятия есть простой физический смысл. Представим себе граф из шариков (вершин), соединённых нитками одинаковой длины (рёбрами). Потянем граф вверх за вершину s; за ней потянутся все вершины, достижимые из s. Чтобы найти расстояния от них до s, мы можем теперь просто измерить, насколько они ниже s.

Например, на рис. 4.2 расстояние между вершинами S и B равно 2, и есть два кратчайших пути между ними. Когда граф подвешен за S, все рёбра этих

Рис. 4.1. (a) Простой граф и (b) его дерево поиска в глубину.

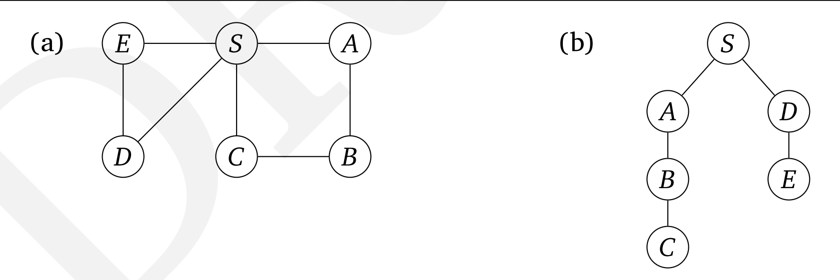
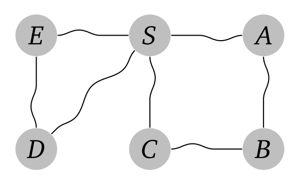
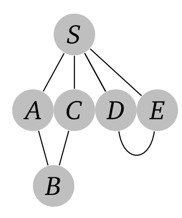


Рис. 4.2. Физическая модель графа.





двух путей натягиваются. А ребро (D, E) провисает, поскольку не входит ни в один из кратчайших путей из вершины S.

4.2. Поиск в ширину

Вершины графа рис. 4.2, подвешенного за S, разбиваются на слои: сама S, вершины на расстоянии 1 от S, вершины на расстоянии 2 и так далее. Можно находить расстояния от S до всех вершин, переходя от уровня к уровню. Когда вершины уровней $0, 1, 2, \ldots, d$ определены, легко найти вершины уровня $d+1$: это просто ещё не просмотренные вершины, смежные с вершинами уровня d. Эти рассуждения наталкивают нас на алгоритм, работающий в каждый момент с двумя уровнями: некоторым уровнем d, который уже полностью известен, и уровнем $d+1$, который находится просмотром соседей вершин уровня d.

Поиск в ширину (breadth-first search, BFS) непосредственно реализует эту простую идею (рис. 4.3). Изначально очередь Q содержит только вершину S, то есть вершину на расстоянии 0. Для каждого последующего расстояния $d = 1, 2, 3, \ldots$ найдётся момент времени, в который Q содержит все вершины на расстоянии d и только их. Когда все эти вершины будут обработаны (извлечены из очереди), их непросмотренные соседи окажутся добавленными в конец очереди, то есть мы перейдём к следующему значению d.

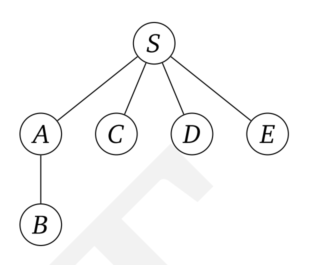
Запустим этот алгоритм для вершины S в графе рис. 4.1. Считаем, что соседи каждой вершины обрабатываются в алфавитном порядке. На рис. 4.4 слева показан порядок обхода вершин, а справа изображено дерево поиска в ширину. Оно содержит только рёбра, при переходе по которым обнаруживались новые вершины. В отличие от дерева поиска в глубину все пути данного дерева с началом в S являются кратчайшими. Поэтому оно называется *деревом кратчайших путей* (shortest-path tree).

Рис. 4.3. Поиск в ширину.

```
процедура BFS(G, s)
{Вход: граф G(V, E), вершина s ∈ V.}
{Выход: для всех вершин u, достижимых из s,
  dist[u] будет равно расстоянию от s до u.}
для всех вершин u ∈ V:
  dist[u] ← ∞
dist[s] ← 0
Q ← {s} {очередь из одного элемента}
пока Q не пусто:
  u ← EJECT(Q)
  для всех рёбер (u, v) ∈ E:
    если dist[v] = ∞:
      INJECT(Q, v)
      dist[v] ← dist[u] + 1
```

Рис. 4.4. Результат поиска в ширину для графа рис. 4.1.

порядок посещения	содержание очереди после обработки вершины
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[ ]



Этот алгоритм можно применять и к ориентированным графам. В них тоже можно определить расстояние от вершины S до вершины T как минимальное число рёбер, которое надо пройти (в их направлении), чтобы из S попасть в T. Расстояние при этом уже не будет симметрично, но понятие кратчайшего пути имеет смысл, и наш алгоритм по-прежнему годится.

Корректность и время работы

Убедимся в корректности алгоритма. Мы утверждаем, что

> для всех $d = 0, 1, 2, \ldots$ найдётся момент времени, когда: 1) для всех вершин на расстоянии не более d это расстояние уже помещено в массив `dist`; 2) для всех оставшихся вершин значения в `dist` равны ∞; 3) очередь содержит в точности вершины на расстоянии d.

Данное утверждение легко доказывается по индукции (проверьте).

Как и в случае поиска в глубину, время работы поиска в ширину линейно, то есть равно $O(|V| + |E|)$. Действительно, каждая вершина помещается в очередь ровно один раз — при её обнаружении. Поэтому общее количество операций с очередью есть $2|V|$. Вся оставшаяся работа производится во внутреннем цикле алгоритма. На это требуется время $O(|E|)$, поскольку каждое ребро в данном цикле просматривается один раз (для ориентированных графов) или два раза (для неориентированных).

Теперь, когда мы знаем и поиск в глубину, и поиск в ширину, интересно сравнить их способы обхода графа. Поиск в глубину стремится вперёд, и возвращается назад, чтобы пойти вбок, только если впереди уже нет новых вершин. Поэтому он может долго добираться до вершины, которая находится совсем близко (рис. 4.1). Поиск в ширину обходит вершины в порядке увеличения расстояний до них от начальной вершины, как фронт волны на воде.

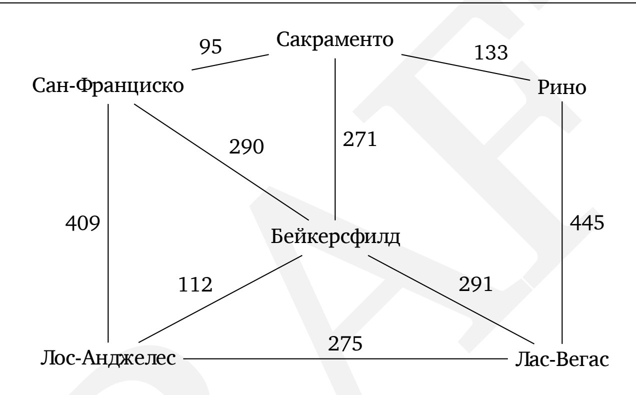
Если в алгоритме поиска в ширину заменить очередь на стек (вынимается первым тот элемент, который положен последним), то он превратится в поиск в глубину (в нерекурсивном варианте).

При поиске в ширину нас интересуют расстояния от s, поэтому мы не перезапускаем поиск в других связных компонентах (недоступные вершины просто игнорируются).

4.3. Длины рёбер

В алгоритме поиска в ширину мы считали все рёбра одинаково длинными. На практике это обычно не так. Например, если вы едете из Сан-Франциско в Лас-Вегас и хотите найти оптимальный путь, надо учитывать длины рёбер в дорожном графе (рис. 4.5). В данном разделе мы будем считать, что каждому ребру графа $e \in E$ приписана его «длина» l_e. Для $e = (u, v)$ это число мы будем также обозначать через $l(u, v)$ или l_{uv}. Реально это может быть длина в километрах, или время в часах, или цена билета в рублях (в зависимости от того, что мы оптимизируем). В некоторых ситуациях оказываются полезными даже отрицательные длины.

Рис. 4.5. Дорожный граф

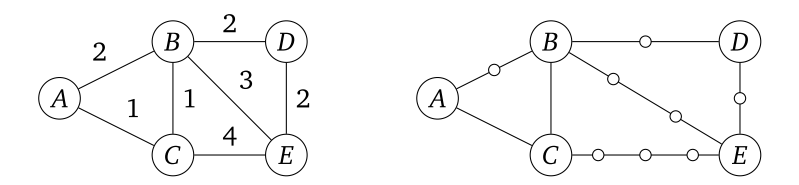


4.4. Алгоритм Дейкстры

4.4.1. Модификация поиска в ширину

Пусть длины всех рёбер — натуральные числа. Тогда задачу поиска кратчайшего пути можно свести к случаю рёбер длины 1, расставив на всех дорогах «верстовые столбы» и объявив их новыми вершинами (соответственно разбиваются и рёбра). Старые вершины и расстояния между ними останутся без изменений, так что можно найти эти расстояния, запустив поиск в ширину на новом графе (рис. 4.6).

Рис. 4.6. Разбиение рёбер на единичные куски.

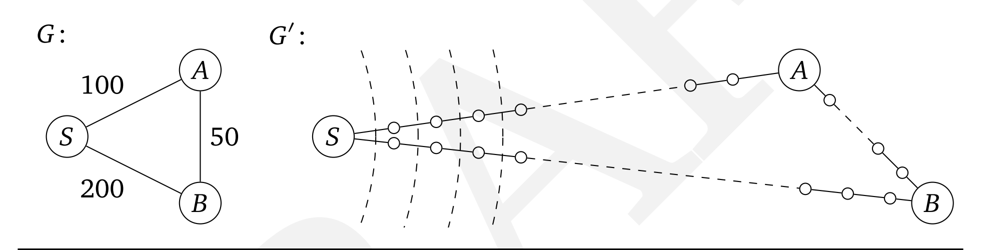


Будильники

Очевидный недостаток такого решения состоит в том, что при длинных рёбрах нужно много вспомогательных вершин, и алгоритм будет тратить много времени, двигаясь от одного столба до другого.

Для примера рассмотрим граф G и граф G' с добавленными вершинами, показанные на рис. 4.7. Запустим из S поиск в ширину (со скоростью ребро за минуту). Тогда первые 99 минут он будет двигаться вдоль S–A и S–B от столба к столбу. На это и смотреть скучно, так что мы лучше поспим до того момента, когда произойдёт что-то интересное (мы дойдём до настоящей вершины). Поставим два будильника: один на время 100 для вершины A, второй — на 200 для B. Первый сработает раньше: поиск дойдёт сначала до вершины A. Из A начнётся новое движение, которое дойдёт до B в момент 150 (раньше, чем движение из S).

Рис. 4.7. Поиск в ширину на новом графе. Пунктирные линии показывают начальные «фронты волн».



Опишем ситуацию в общем виде: поиск в ширину продвигается по рёбрам графа G, и для каждой вершины установлен будильник на ожидаемое время прибытия уже запущенного движения. (Реально поиск может прийти туда и раньше, пройдя через другую вершину, как это было в примере.) Главное: *пока не сработает хотя бы один будильник, ничего интересного не будет* (движение будет продолжаться по рёбрам). Звонок будильника означает, что поиск в ширину добрался до некоторой вершины исходного графа $u \in V$. В этот момент начинаются движения по рёбрам, исходящим из u, и это надо учесть в их концевых вершинах и перезавести будильники на концах этих рёбер (если новое время прибытия раньше прежнего). Вот схема алгоритма:

- Завести будильник для s на время 0.
- Пока есть заведённые, но не сработавшие будильники:
  — Взять самый ранний из них (пусть он в вершине u на момент T).
  — Констатировать, что расстояние от s до u равно T.
  — Для каждой вершины v, смежной с u в G:
  если для вершины v будильник ещё не заведён, завести его на время $T + l(u, v)$;
  если будильник заведён на время, большее чем $T + l(u, v)$, то переставить его на это меньшее время (а иначе оставить как было).

Алгоритм Дейкстры

Алгоритм с будильниками находит кратчайшие пути в графах с положительными целыми весами. Он практически готов к использованию, осталось только как-нибудь реализовать механизм будильников. Подходящая для этих целей структура данных называется *очередью с приоритетами*; обычно её реализуют с помощью *кучи*. Очередь с приоритетами хранит множество элементов (в нашем случае — вершин графа) и соответствующих им числовых значений — ключей (ожидаемое время, в которое прозвонит будильник) и выполняет следующие операции:

Insert: добавить новый элемент (с указанным ключом) в очередь.

DecreaseKey: уменьшить ключ, соответствующий заданному элементу очереди, до заданного значения.

DeleteMin: выдать элемент с минимальным ключом, удалив его из очереди.

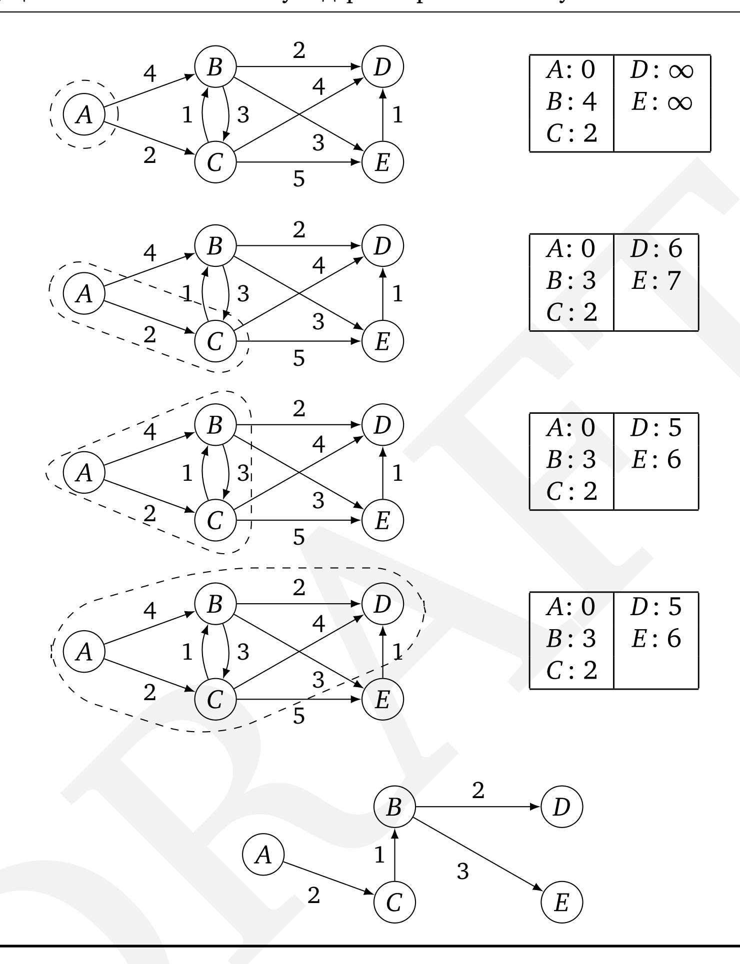
MakeQueue: построить очередь из заданных элементов с ключами (выделяется в особую операцию, поскольку иногда это быстрее, чем добавлять все элементы по очереди).

С помощью первых двух операций мы будем ставить и переставлять будильники, а с помощью третьей — узнавать, какой из будильников прозвонит следующим. Всё вместе даёт *алгоритм Дейкстры* (D>kstra) поиска кратчайших путей (рисунок 4.8).

Рис. 4.8. Алгоритм Дейкстры для нахождения кратчайших путей.

```
процедура Dijkstra(G, l, s)
{Вход: граф G(V, E) (ориентированный или нет) с неотрицательными
  длинами рёбер {l_e: e ∈ E}; вершина s ∈ V.}
{Выход: для всех вершин u, достижимых из s,
  dist[u] будет равно расстоянию от s до u (и ∞ для недостижимых).}
для всех вершин u ∈ V:
  dist[u] ← ∞
  prev[u] ← nil
dist[s] ← 0
H ← MakeQueue(V) {в качестве ключей используются значения dist}
пока H не пусто:
  u ← DeleteMin(H)
  для всех рёбер (u, v) ∈ E:
    если dist[v] > dist[u] + l(u, v):
      dist[v] ← dist[u] + l(u, v)
      prev[v] ← u
      DecreaseKey(H, v, dist[v])
```

Рис. 4.9. Алгоритм Дейкстры для начальной вершины A; для каждой итерации показаны текущие dist-значения. Внизу — дерево кратчайших путей.



Через dist[u] обозначено время, на которое заведён будильник в вершине u. Если оно равно ∞, значит, будильник для этой вершины ещё не заводили.[1]

Мы также добавили в алгоритм массив prev: для каждой вершины u мы помним, откуда в неё ведёт кратчайший (обнаруженный на данный момент) путь, и по этим ссылкам можем пройти этот путь в обратную сторону. На рис. 4.9 показан пример работы алгоритма и построенное им дерево кратчайших путей.

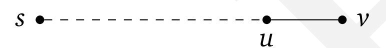
[1] Отличие от словесных объяснений выше: мы сразу же помещаем в очередь будильники для всех вершин, в том числе и ещё не заведённые (ключ ∞). Пустой очереди в нашем предыдущем описании соответствует ситуация, когда все оставшиеся в очереди вершины имеют бесконечный ключ. Они будут одна за другой изъяты из очереди, но не потребуют изменений ключей. — *Прим. ред.*

Мы построили алгоритм Дейкстры как сокращённую версию поиска в ширину в графе с добавленными вершинами. Но тот же алгоритм можно объяснить и другим способом.

4.4.2. Другое объяснение

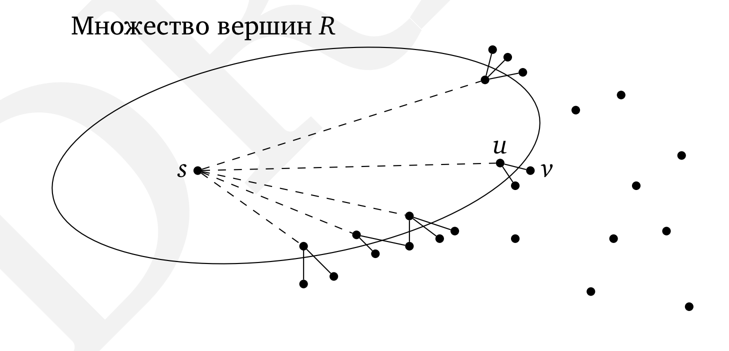
Начав с вершины s, будем последовательно расширять множество R вершин, до которых уже найдены расстояния и кратчайшие пути. Добавлять вершины в это множество будем в порядке увеличения расстояний до них. Следующей нужно добавить вершину v, расстояние от s до которой минимально среди вершин, не принадлежащих R. Как найти такую v?

Представим себе кратчайший путь из s в эту самую v; пусть u — предпоследняя вершина. Если u не в R, то почему бы нам не взять u вместо v? ведь вершина u не дальше от s, чем v. (Мы предполагаем, что все длины неотрицательны.) Вообще можно заменить v на первую вершину не из R вдоль этого пути; поскольку длины неотрицательны, расстояние не увеличится (а при положительных длинах — уменьшится):



Вывод: можно найти подходящую вершину v, сравнивая пути, которые остаются в R до последнего момента и лишь на последнем шаге выходят из R (рис. 4.10). Из таких путей надо выбрать самый короткий. Вершину, в которую он ведёт, и надо добавить в R.

Рис. 4.10. Удлинение уже найденных кратчайших путей на одно ребро.



Путей этих много. Дело упрощается тем, что, во-первых, расстояния до вершин в R мы уже знаем и нужно лишь добавить последнее ребро в пути, а во-вторых, когда в R добавляется новая вершина v, класс интересующих нас путей пополняется лишь путями, которые на предпоследнем шаге проходят через v. Это позволяет не просматривать все пути снова, а лишь корректировать прошлый минимум с учётом новых путей. Получаем такую схему ал-

горитма, где через dist[v] обозначена текущая длина кратчайшего из однорёберных расширений, ведущих в v (для вершин, не являющихся соседями вершин из R, значение dist[v] равно ∞):

```
положить dist[s] ← 0, а все остальные dist[·] — равными ∞
R ← {} {множество вершин с уже найденными расстояниями}
пока R ≠ V :
  выбрать вершину v ∉ R с минимальным значением dist[v]
  добавить v к R
  для всех рёбер (v, z) ∈ E:
    для dist[z] > dist[v] + l(v, z):
      dist[z] ← dist[v] + l(v, z)
```

Добавление очереди с приоритетами (с помощью которой выбирается вершина с минимальным значением) даёт уже известный нам алгоритм Дейкстры (рис. 4.8). Его корректность можно доказать по индукции. Точнее, надо доказывать по индукции, что после любого числа итераций внешнего цикла

(1) найдётся значение d, для которого все вершины множества R находятся на расстоянии не более d от s, а все остальные вершины — на расстоянии хотя бы d;
(2) для любой вершины u значение dist[u] равно длине кратчайшего пути от s до u, все вершины которого, не считая последней, лежат в R (если такого пути нет, то dist[u] $= \infty$).

Мы начинали обсуждение алгоритма Дейкстры с добавления промежуточных вершин и предполагали тогда, что длины рёбер целые. Теперь это предположение уже никак не используется.

4.4.3. Время работы

Как видно из рис. 4.8, алгоритм Дейкстры очень похож на поиск в ширину, но работает он дольше: действия с очередью с приоритетами сложнее, чем операции Inject и Eject для обычной очереди. Оценивая время работы, заметим, что для создания очереди с приоритетами достаточно последовательно добавить в неё все элементы. Поэтому операция MakeQueue работает не дольше, чем $|V|$ операций Insert. Остаётся оценить время работы $|V|$ операций DeleteMin и $|V| + |E|$ операций Insert/DecreaseKey. Оценка, конечно же, зависит от конкретной реализации очереди с приоритетами, а они бывают разные, как мы увидим в следующем разделе. Например, реализация с помощью двоичной кучи даёт оценку $O((|V| + |E|) \log |V|)$.

4.5. Реализации очередей с приоритетами

4.5.1. Массив

Самое простое — хранить ключи в неупорядоченном массиве ключей для всех возможных элементов (вершин графа в алгоритме Дейкстры), как-то отмечая удалённые.

Операции Insert и DecreaseKey работают быстро, за время $O(1)$, так как затрагивают только один элемент. В то же время DeleteMin требует линейного времени работы: для поиска минимума надо пройти весь массив.

4.5.2. Двоичная куча

В двоичной куче элементы хранятся в двоичном дереве, где вершины заполняют уровни сверху вниз и (на одном уровне) слева направо. Хранимые ключи при этом должны обладать таким свойством: *ключ в любой вершине дерева не превышает значения ключей его детей* (тех, которые есть в дереве). Отсюда следует, что корень дерева всегда содержит наименьший ключ. Пример двоичной кучи см. на рис. 4.11(a).

Операция Insert выполняется следующим образом: новый элемент добавляется в первую доступную позицию на нижнем уровне дерева и «всплывает» вверх по дереву. Это означает, что пока этот элемент «легче» (меньше) своего родителя, он меняется с родителем местами (рис. 4.11(b)—(d)). Количество обменов не превышает высоты дерева, равной $\lfloor \log_2 n \rfloor$, где n — количество элементов в дереве. Подобным же образом работает операция DecreaseKey, за исключением того, что элемент уже находится в дереве, и ему нужно лишь «всплыть» после уменьшения.

Операция DeleteMin забирает элемент из корня дерева. Образовавшееся пустое место заполняется элементом из последней вершины, который затем «тонет» (пока он больше хотя бы одного из детей, он меняется местами с меньшим из детей, см. рис. 4.11(b)—(d)). Эта операция также требует времени $O(\log n)$.

Заполнение дерева сверху вниз и слева направо позволяет хранить его элементы подряд в указанном порядке. Если нумеровать с единицы, то детьми вершины j будут $2j$ и $2j+1$, а родителем будет $\lfloor j/2 \rfloor$ (упражнение 4.16).

4.5.3. d-ичная куча

В двоичной куче у каждой вершины до двух детей; в d-ичной — до d. Это сокращает высоту дерева для n элементов до $\Theta(\log_d n) = \Theta((\log n)/(\log d))$. Операция Insert ускоряется в $\Theta(\log d)$ раз, но DeleteMin замедляется до $O(d \log_d n)$. (Понятно ли, в чём тут дело?)

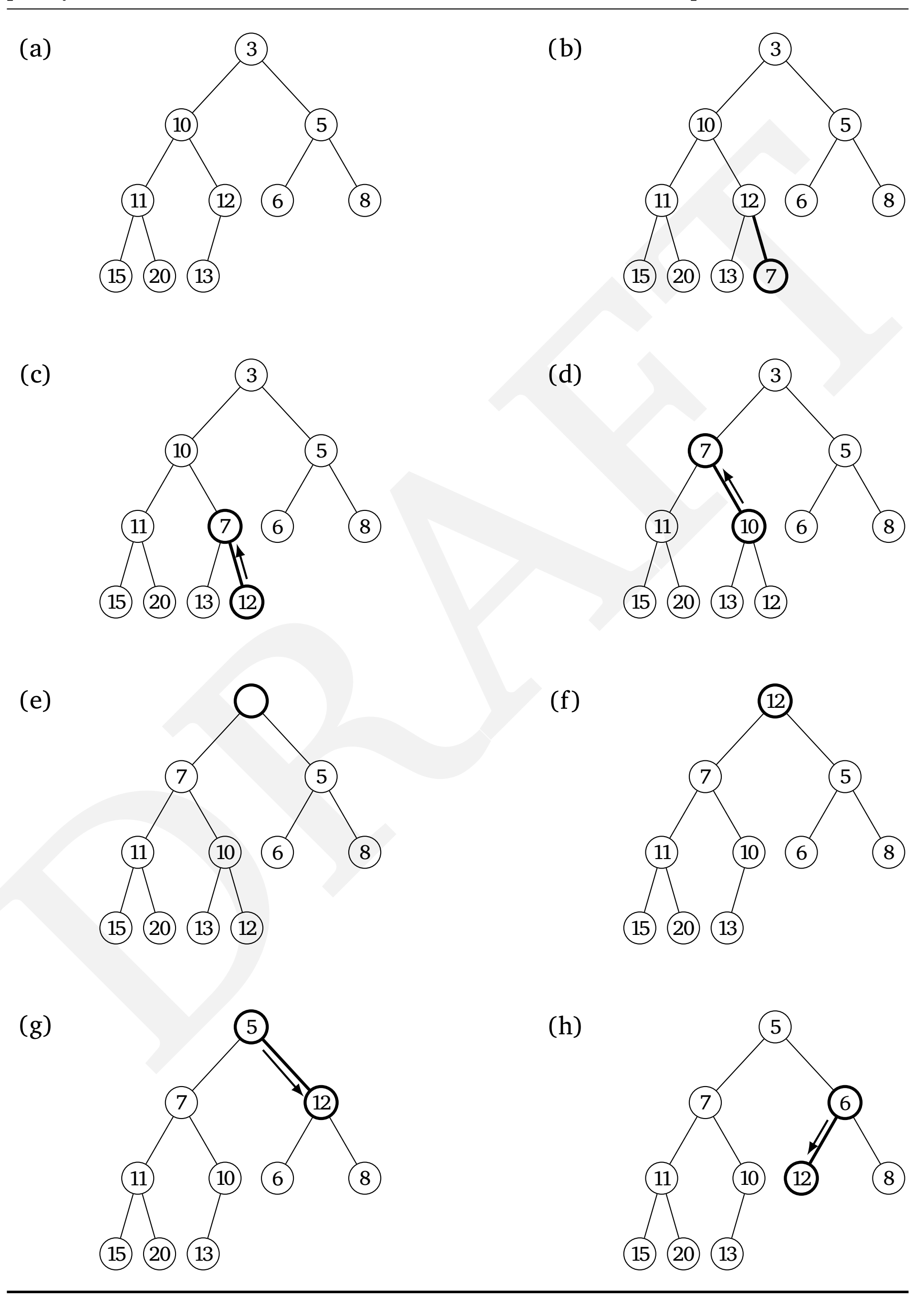
Такую кучу столь же легко хранить в массиве. Родителем вершины j является вершина $\lceil (j-1)/d \rceil$, а детьми — вершины $(j-1)d+2, \ldots, (j-1)d + (d+1) = jd+1$ (те из них, которые не больше n, см. упражнение 4.16).

4.6. Кратчайшие пути и отрицательные веса

4.6.1. Рёбра отрицательного веса

Алгоритм Дейкстры основан на том, что на кратчайшем пути от начальной вершины s до любой вершины v все промежуточные вершины не дальше от начала, чем v. Это условие может нарушаться, если в графе есть рёбра отри-

Рис. 4.11. (a) Двоичная куча из 10 элементов. Показаны только значения ключей. (b)—(d) Ключ 7 добавляется вниз и всплывает. (e)—(h) При операции DeleteMin корень удаляется, а место заполняется последним элементом, который затем тонет.



Какую реализацию очереди с приоритетами выбрать?

Время работы алгоритма Дейкстры сильно зависит от выбранной реализации очереди с приоритетами. Ниже приведены возможные варианты.

Реализация	DeleteMin	Insert/ DecreaseKey	$\lvert V\rvert \times$ DeleteMin ($\lvert V\rvert + \lvert E\rvert$) $\times$ Insert
Массив	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Двоичная куча	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert)\log \lvert V\rvert)$
d-ичная куча	$O\left(\frac{d \log \lvert V\rvert}{\log d}\right)$	$O\left(\frac{\log \lvert V\rvert}{\log d}\right)$	$O\left((\lvert V\rvert \cdot d + \lvert E\rvert)\frac{\log \lvert V\rvert}{\log d}\right)$
Куча Фибоначчи	$O(\log \lvert V\rvert)$	$O(1)$ (аморт.)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

Даже простейшая реализация очереди с приоритетами на базе массива даёт приемлемое время $O(|V|^2)$, а при использовании двоичной кучи получится $O((|V|+|E|)\log|V|)$. Что предпочтительнее?

Всё зависит от того, является граф *разреженным* (число рёбер достаточно мало) или *плотным* (число рёбер достаточно велико). Максимальное $|E|$ (в полном графе) имеет порядок $|V|^2$. Если $|E|$ близко к максимуму, реализация на базе массива будет быстрее. Двоичная куча становится предпочтительнее, если $|E|$ заметно меньше $|V|^2/\log|V|$.

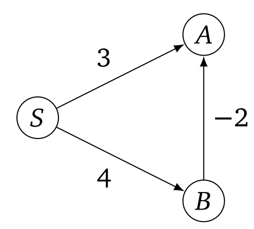
По аналогии с двоичной кучей можно рассмотреть d-ичную. Время работы для такой кучи зависит от d, и оптимальным значением d будет *средняя степень* графа, то есть $d \approx |E|/|V|$. Подбирая d, можно получать эффективный алгоритм и для разреженных, и для плотных графов. Время работы на разреженных графах, в которых $|E| = O(|V|)$, составляет $O(|V|\log|V|)$, как и для двоичной кучи. В случае плотных графов, когда $|E| = \Omega(|V|^2)$, время работы, как и при использовании массива, составляет $O(|V|^2)$. Кроме того, для графов промежуточной плотности $|E| = |V|^{1+\delta}$ время работы составляет $O(|E|)$, а это линейное время!

В последней строчке таблицы показано время работы для продвинутой структуры данных, именуемой *кучей Фибоначчи*. Несмотря на впечатляющие оценки, она сложна в реализации, что снижает её практическую привлекательность, и мы лишь упомянем любопытную особенность. Последовательные вызовы операции Insert для такой кучи выполняются за разное время, и некоторые могут быть долгими, но *среднее* (average) время работы этой операции на протяжении работы алгоритма обязательно будет $O(1)$. В таких случаях (с одним из которых мы столкнёмся в главе 5) говорят, что *амортизированная* (amortized) стоимость операции Insert равна $O(1)$.

цательного веса. На рис. 4.12 кратчайший путь из вершины S в вершину A проходит через вершину B, расположенную дальше, чем сама A.

Чтобы понять, как преодолеть эту трудность, вспомним общую идею алгоритма Дейкстры. Алгоритм поддерживает следующий важный инвариант:

Рис. 4.12. Алгоритм Дейкстры не будет работать при наличии рёбер отрицательного веса.



значения dist[·] в любой момент не меньше истинных. Изначально они устанавливаются в ∞ и обновляются при обнаружении новых рёбер:

процедура Update($(u, v) \in E$)
dist[v] $\leftarrow$ min{dist[v], dist[u] + $l(u, v)$}

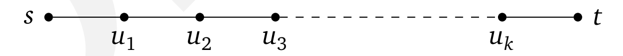
Операция Update отражает тот факт, что расстояние до вершины v не может превышать расстояния до u плюс $l(u, v)$. При этом:

1. Update находит истинное расстояние до вершины v в том случае, когда u — предпоследняя вершина в кратчайшем пути до v и dist[u] уже установлено правильно.

2. dist[v] никогда не станет меньше, чем кратчайшее расстояние от s до v («безопасность»).

Таким образом, вызовов процедуры Update может быть недостаточно, но не может быть много. Алгоритм Дейкстры указывает некоторый конкретный способ произвести достаточное количество вызовов Update. Для графов с отрицательными весами делаемых им вызовов может не хватить. Но, может быть, можно найти нужную последовательность вызовов как-то иначе?

Пусть, скажем, дан кратчайший путь из s в какую-то вершину t. Этот путь может состоять не более чем из $|V| - 1$ ребра. (Почему?)



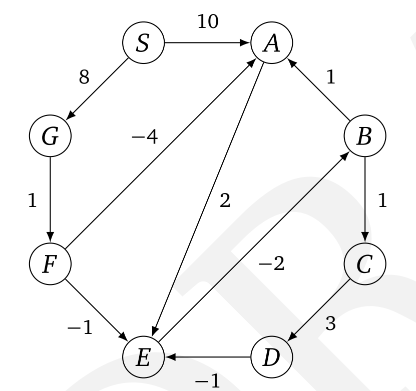
Если среди операций Update есть $(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t)$ *в таком порядке*, то по первому свойству расстояние до t будет вычислено верно. Не имеет значения, идут ли эти обновления подряд или чередуются с чем-то ещё; неважно, что происходит в остальной части графа, потому что обновления *безопасны*.

Правда, мы не знаем заранее кратчайшие пути, как мы сможем обновить нужные рёбра в правильном порядке? Вот простое решение: сделать $|V| - 1$ шагов, обновляя на каждом шаге все рёбра по одному разу! Полученная процедура называется алгоритмом Беллмана—Форда и имеет время работы $O(|V| \cdot |E|)$. Алгоритм приведён на рис. 4.13, а пример работы — на рис. 4.14.

Рис. 4.13. Алгоритм Беллмана—Форда для нахождения кратчайших путей из одной вершины.

```
процедура ShortestPaths(G, l, s)
{Вход: ориентированный граф G(V, E) без циклов отрицательного веса,
  длины рёбер {l_e : e ∈ E}, вершина s ∈ V.}
{Выход: для всех вершин u, достижимых из s,
  dist[u] будет равно расстоянию от s до u.}
для всех вершин u ∈ V:
  dist[u] ← ∞
  prev[u] ← nil
dist[s] ← 0
повторить |V| − 1 раз:
  для всех рёбер e ∈ E:
    Update(e)
```

Рис. 4.14. Пример работы алгоритма Беллмана—Форда.



	Итерация							
Вершина	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Замечание по поводу реализации: во многих графах максимальное количество рёбер в любом кратчайшем пути значительно меньше $|V|$, так что достаточно меньшего числа обновлений. Поэтому стоит добавить в алгоритм дополнительную проверку, чтобы завершить его, как только при очередном проходе по рёбрам не произойдёт ни одного изменения.

4.6.2. Циклы отрицательного веса

Если длину ребра (E, B) на рис. 4.14 сделать равной -4, в графе появится *цикл отрицательного веса* $A \to E \to B \to A$. В таких случаях вопрос о кратчайших путях лишается смысла. Из A в E существует путь веса 2. Однако при проходе по циклу появится также путь веса 1, а пройдя по циклу несколько раз, мы получим веса 0, −1, −2 и так далее.

Задача о кратчайших путях в такой ситуации не имеет смысла, так что и наш алгоритм из раздела 4.6.1 не будет правильно работать. Но где именно он опирается на отсутствие отрицательных циклов? В том месте, где мы сочли, что кратчайший путь из s в t *существует* и состоит не более чем из $|V|-1$ ребра.

К счастью, наличие цикла отрицательного веса нетрудно обнаружить (и выдать предупреждение). В самом деле, если такой цикл есть (и доступен из начальной вершины: недоступные не играют роли), то каждый следующий UPDATE-раунд приведёт к уменьшению какого-то из значений `dist`. Поэтому после $|V|-1$ проходов достаточно сделать ещё один контрольный проход и проверить, что при этом ничего не поменялось. Действительно, если на $|V|$-м проходе что-то изменилось, то в графе есть цикл отрицательного веса (мы доказали именно это, установив, что в графе без циклов отрицательного веса алгоритм найдёт все кратчайшие пути за $|V|-1$ проход). Обратно, допустим, что в графе есть цикл отрицательного веса, доступный из начальной вершины. В какой-то момент значения $\mathtt{dist}[u]$ для вершин цикла станут конечными. Если после этого вызов UPDATE(u, v) не меняет значение $\mathtt{dist}[v]$, то $\mathtt{dist}[v] \leqslant \mathtt{dist}[u] + l(u, v)$. Пусть на очередном раунде значение `dist` не было изменено ни для одного ребра; просуммировав такие неравенства для всех рёбер цикла, получим противоречие с тем, что вес цикла отрицателен.

4.7. Кратчайшие пути в ациклических графах

Есть два типа графов, в которых заведомо нет циклов отрицательного веса: графы без отрицательных рёбер и графы без циклов. Мы уже знаем, как можно эффективно обрабатывать графы первого типа. Для второго типа (ацикли-

Рис. 4.15. Алгоритм нахождения кратчайших путей из одной вершины для ориентированных ациклических графов.

```
процедура DAGShortestPaths(G, l, s)
{Вход: ориентированный ациклический граф G(V, E),
  длины рёбер {l_e: e ∈ E}, вершина s ∈ V.}
{Выход: для всех вершин u, достижимых из s,
  dist[u] будет равно расстоянию от s до u.}
для всех вершин u ∈ V:
  dist[u] ← ∞
  prev[u] ← nil
dist[s] ← 0
топологически упорядочить G
для всех u ∈ V в найденном порядке:
  для всех рёбер (u, v) ∈ E:
    Update(u, v)
```

ческих ориентированных графов) задача поиска кратчайших путей из данной вершины может быть решена ещё быстрее, за линейное время.

Как и раньше, нам нужно произвести последовательность обновлений, которая покрывает (в правильном порядке) любой кратчайший путь. Вспомним теперь, что

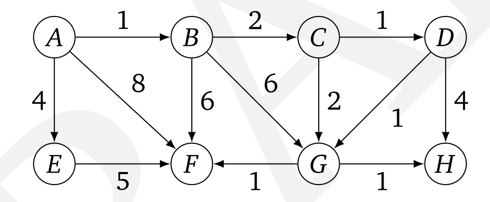
> в ориентированном ациклическом графе вершины в любом пути сохраняют порядок после топологической сортировки.

Поэтому достаточно топологически упорядочить граф (поиском в глубину), после чего пройтись по вершинам в найденном порядке, обновляя исходящие рёбра (рис. 4.15).

Обратите внимание на то, что мы нигде не используем тот факт, что веса положительны. В частности, тот же самый алгоритм позволяет найти *самый длинный путь* в ориентированном ациклическом графе, просто поменяв знаки весов всех рёбер.

Упражнения

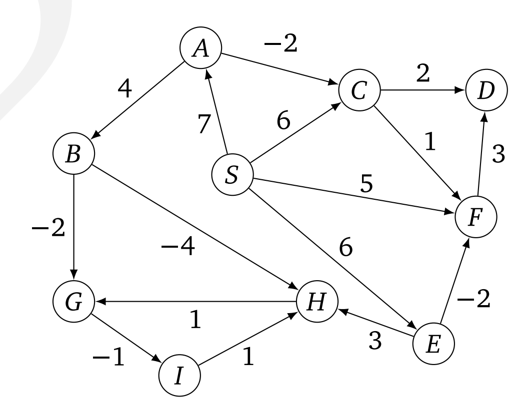
4.1. Примените алгоритм Дейкстры к графу на рисунке и вершине A.



(a) Заполните таблицу, содержащую промежуточные расстояния до всех вершин на каждой итерации алгоритма.

(b) Нарисуйте полученное дерево кратчайших путей.

4.2. То же самое, но теперь с алгоритмом Беллмана—Форда и для графа с отрицательными весами:



4.3. *Квадраты.* Постройте алгоритм, определяющий, содержит ли данный неориентированный граф $G = (V, E)$ простой цикл (то есть цикл без самопересечений) длины 4. Время работы алгоритма должно быть $O(|V|^3)$. Можно считать, что граф задан списком смежности или матрицей смежности (как вам проще).

4.4. Рассмотрим следующий способ нахождения кратчайшего цикла в неориентированном графе с рёбрами единичной длины.

> Найденное во время поиска в глубину обратное ребро (v, w) вместе с путём от w до v в дереве поиска в глубину образует цикл. Длина такого цикла равна $\texttt{level}[v] - \texttt{level}[w] + 1$, где level обозначает уровень в дереве (расстояние от корня). Попробуем действовать так:
>
> - Запустить поиск в глубину и в процессе работы вычислять уровни вершин.
> - Каждый раз при нахождении обратного ребра вычислять длину получившегося цикла и запоминать её, если она меньше всех предыдущих длин.

Приведите пример, где этот алгоритм не находит кратчайший цикл, и коротко объясните причину.

4.5. В графе может быть несколько кратчайших путей между какими-то вершинами. Постройте линейный по времени алгоритм, находящий в заданном неориентированном графе с единичными весами количество различных кратчайших путей между заданными двумя вершинами.

4.6. Докажите, что рёбра $\{u, \texttt{prev}[u]\}$, полученные в результате работы алгоритма Дейкстры, образуют дерево.

4.7. Дан ориентированный граф $G = (V, E)$ с (возможно, отрицательными) весами на рёбрах, а также $s \in V$ и дерево $T = (V, E')$, $E' \subseteq E$. Постройте алгоритм, проверяющий за линейное время, является ли T деревом кратчайших путей из s.

4.8. Профессор О. П. Рометчивый предлагает следующий способ нахождения кратчайшего пути из s в t в данном ориентированном графе, содержащем рёбра отрицательного веса. Прибавим достаточно большую константу к весам всех рёбер и сделаем все веса положительными, после чего воспользуемся алгоритмом Дейкстры.

Корректен ли такой подход? Если да, то докажите это, если нет — укажите контрпример.

4.9. Рассмотрим ориентированный граф, в котором рёбра отрицательного веса выходят только из вершины s, но циклов отрицательного веса при этом нет. Может ли алгоритм Дейкстры, вызванный для вершины s, выдать неправильный ответ в данном случае?

4.10. Дан ориентированный граф с (возможно, отрицательными) весами на рёбрах. Известно, что для двух его вершин найдётся кратчайший путь из пер-

вой во вторую не более чем из k рёбер. Постройте алгоритм, находящий этот путь за время $O(k|E|)$.

4.11. Постройте алгоритм, который по данному ориентированному графу с положительными весами находит длину его самого короткого цикла (если в графе циклов нет, алгоритм должен сообщить об этом). Время работы алгоритма должно быть $O(|V|^3)$.

4.12. Постройте алгоритм, который по данному ориентированному графу $G=(V,E)$ с положительными весами и его ребру находит длину самого короткого цикла, содержащего это ребро, за время $O(|V|^2)$.

4.13. Пусть неориентированный граф $G=(V,E)$ задаёт множество городов и дорог между ними. Для каждой дороги $e\in E$ известна её длина в километрах l_e. Предположим, что вам необходимо добраться из города s в город t, но ваша машина может проехать без дозаправки не больше L километров. Заправки есть во всех городах, но не вдоль дорог (так что годятся только маршруты, где все рёбра не длиннее L).

(a) Постройте алгоритм, определяющий за линейное время, можно ли при таких ограничениях добраться из s в t.

(b) Допустим теперь, что вы хотите купить новую машину. Какой минимальный объём бака машины позволит добраться из s в t? Постройте алгоритм, дающий ответ за время $O((|V|+|E|)\log|V|)$.

4.14. Дан сильно связный ориентированный граф $G=(V,E)$ с положительными весами и вершина $v_0\in V$. Постройте эффективный алгоритм, который для всех пар вершин находит кратчайшие среди путей, проходящих через v_0.

4.15. Между двумя вершинами графа может быть несколько разных путей минимальной длины. Пусть дан граф $G=(V,E)$ с положительными весами и его вершина $s\in V$. Найдите все вершины t, для которых есть лишь один кратчайший путь из s в t, за время $O((|V|+|E|)\log|V|)$. (Вершина s входит в их число.)

4.16. В разделе 4.5.2 описан способ хранения двоичного дерева с n вершинами в массиве с индексами $1, 2, \ldots, n$.

(a) Покажите, что родитель j-го элемента имеет номер $\lfloor j/2\rfloor$, а его дети — номера $2j$ и $2j+1$ (если эти числа не больше n).

(b) Как вычислить соответствующие номера, если в массиве хранится не двоичное, а d-ичное дерево?

На рис. 4.16 мы, следуя Р. Тарьяну[1], приводим алгоритм, реализующий двоичную кучу. Она хранится в массиве h, который дополнительно поддерживает две следующие операции за время $O(1)$:

- $|h|$ возвращает текущее количество элементов в массиве;
- h^{-1} возвращает индекс заданного элемента в массиве.

[1]См. *R. E. Tarjan*. Data Structures and Network Algorithms. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

Рис. 4.16. Операции с двоичной кучей.

```
процедура Insert(h, x)
BubbleUp(h, x, |h| + 1)

процедура DecreaseKey(h, x)
BubbleUp(h, x, h^{-1}(x))

функция DeleteMin(h)
если |h| = 0:
  вернуть null
иначе:
  x ← h[1]
  SiftDown(h, h[|h|], 1)
  вернуть x

функция MakeHeap(S)
h ← пустой массив размера |S|
для x ∈ S:
  h[|h| + 1] ← x
для i от |S| до 1:
  SiftDown(h, h[i], i)
вернуть h

процедура BubbleUp(h, x, i)
{помещает элемент x в позицию i массива h и даёт ему «всплыть»}
p ← ⌈i/2⌉
пока i ≠ 1 и key(h[p]) > key(x):
  h[i] ← h[p]; i ← p; p ← ⌈i/2⌉
h[i] ← x

процедура SiftDown(h, x, i)
{помещает элемент x в позицию i массива h и даёт ему «утонуть»}
c ← MinChild(h, i)
пока c ≠ 0 и key(h[c]) > key(x):
  h[i] ← h[c]; i ← c; c ← MinChild(h, i)
h[i] ← x

функция MinChild(h, i)
если 2i > |h|:
  вернуть 0 {нет детей}
иначе:
  вернуть arg min{key(h[j]): 2i ⩽ j ⩽ min{|h|, 2i + 1}}
```

Последнего легко добиться, поддерживая значения h^{-1} во вспомогательном массиве.

(c) Покажите, что время работы процедуры MakeHeap на множестве из n элементов есть $O(n)$. Для какого множества время работы будет наихудшим? (Подсказка: для начала покажите, что время работы не превосходит $\sum_{i=1}^{n} \log(n/i)$.)

(d) Что надо изменить, чтобы хранить d-ичные кучи?

4.17. Допустим, что нам нужно найти кратчайшие расстояния от данной вершины в графе с весами рёбер в интервале $0, 1, \ldots, W$, где W — сравнительно небольшое число.

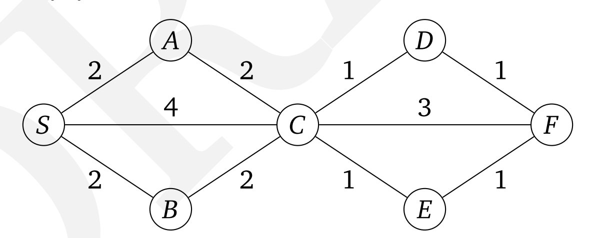
(a) Покажите, что тогда алгоритм Дейкстры можно модифицировать, получив время работы $O(W|V| + |E|)$.

(b) Постройте другой вариант алгоритма с оценкой $O((|V| + |E|) \log W)$.

4.18. Если в графе с различными длинами рёбер есть несколько кратчайших путей между двумя вершинами, часто выбирают тот, в котором *меньше рёбер*. Например, если вершины обозначают города, а веса рёбер — стоимость перелёта, то из нескольких одинаково дорогих вариантов удобнее маршрут с наименьшим количеством пересадок. С учётом этого зададим массив `best` следующим образом:

$$\texttt{best}[u] = \text{минимальное число рёбер на кратчайшем пути из } s \text{ в } u.$$

Например, для графа на рисунке значениями массива `best` для вершин S, A, B, C, D, E, F будут 0, 1, 1, 1, 2, 2, 3 (соответственно).



Постройте эффективный алгоритм, который по данному графу с положительными весами и по данной вершине s заполняет массив `best`.

4.19. *Обобщённая задача о кратчайших путях.* При маршрутизации пакетов в сети задержки есть не только на линиях, но и в узлах (маршрутизаторах). Поэтому разумно рассмотреть обобщение задачи о кратчайших путях.

Пусть для графа заданы не только длины рёбер $\{l_e : e \in E\}$, но и *стоимости вершин* $\{c_v : v \in V\}$. Определим стоимость пути как сумму длин его рёбер и стоимости всех его вершин. Постройте эффективный алгоритм для следующей задачи.

Вход: ориентированный граф $G = (V, E)$; положительные длины рёбер l_e и стоимости вершин c_v; начальная вершина $s \in V$.

Выход: массив cost, в котором для каждой вершины u значение cost$[u]$ есть минимальная стоимость пути из s в u.

Отметим, что cost$[s]=c_s$.

4.20. Пусть граф $G=(V, E)$ представляет собой множество городов V и множество дорог E между ними. У каждой дороги $e\in E$ есть длина l_e. Решено построить новую дорогу, и список E' содержит пары вершин, между которыми новая дорога может быть построена. Для всех возможных новых дорог $e'\in E'$ известна длина $l_{e'}$. Надо найти дорогу из E', строительство которой приведёт к максимальному уменьшению длины кратчайшего пути между заданными двумя городами s и t в G. Постройте эффективный алгоритм для такой задачи.

4.21. Применения задачи о кратчайших путях можно найти даже в валютных операциях. Пусть через $c_1, c_2, \ldots, c_n$ обозначены различные валюты. Например, c_1 может обозначать рубли, c_2 — доллары, c_3 — фунты. Пусть через $r_{i,j}$ обозначен курс конвертации валюты c_i в c_j: на одну единицу валюты c_i можно купить $r_{i,j}$ единиц валюты c_j. Данные курсы удовлетворяют естественному условию $r_{i,j}\cdot r_{j,i}<1$, которое говорит, что обмен туда-обратно невыгоден.

(a) Постройте эффективный алгоритм, который, зная все $r_{i,j}$, находит наиболее выгодную последовательность обменов для конвертации заданной валюты s в заданную валюту t. (Подсказка: для решения задачи представьте входные данные в виде графа с вещественными весами.)

Под влиянием спроса и предложения курсы валют меняются. Представим, что в какой-то момент нашлись валюты $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$, для которых $r_{i_1,i_2}\cdot r_{i_2,i_3}\ldots r_{i_{k-1},i_k}\cdot r_{i_k,i_1}>1$. Это означает, что обменяв какое-то количество валюты c_{i_1} в соответствии с данной последовательностью, мы в итоге получим больше исходного. (Вряд ли это продлится долго, но обидно упустить такую возможность!)

(b) Постройте эффективный алгоритм, проверяющий наличие такой аномальной ситуации. (Подсказка: используйте граф предыдущего пункта.)

4.22. *Задача о путешествующем пароходе.* Представьте, что у вас есть пароход, который может совершать рейсы между портовыми городами из множества V. Посещая город i, вы зарабатываете p_i рублей, а стоимость перемещения из города i в город j составляет $c_{ij}>0$ рублей. Вы хотите найти циклический маршрут, максимизирующий отношение прибыли к затратам.

Для этого рассмотрим ориентированный граф $G=(V, E)$, где вершинами являются города и каждые две вершины соединены ребром. Определим относительную прибыль цикла C следующим образом:

$$r(C)=\frac{\sum\limits_{(i,j)\in C} p_j}{\sum\limits_{(i,j)\in C} c_{ij}}.$$

Через r^* обозначим максимальную относительную прибыль простого цикла. Значение r^* можно найти двоичным поиском, если мы умеем проверять по данному r, верно ли неравенство $r>r^*$.

Итак, рассмотрим произвольное $r > 0$. Присвоим каждому ребру (i, j) вес $w_{ij} = rc_{ij} - p_j$.

(a) Покажите, что если в этом графе есть цикл отрицательного веса, то $r < r^*$.

(b) Покажите, что если все циклы имеют строго положительный вес, то $r > r^*$.

(c) Постройте эффективный алгоритм, который для заданной точности $\varepsilon > 0$ находит цикл C, для которого $r(C) \geqslant r^* - \varepsilon$. Докажите корректность алгоритма и оцените его время работы в терминах $|V|$, ε и $R = \max_{(i,j)\in E}(p_j/c_{ij})$.

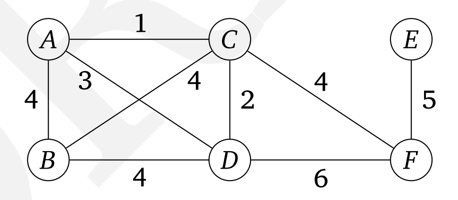
Глава 5

Жадные алгоритмы

Хороший шахматист просматривает варианты на много ходов вперёд и легко выиграет у новичка, который думает только о ближайших последствиях. В других играх разница не так заметна; например, можно неплохо играть в «Эрудит», думая только о ближайшем ходе. Это же относится к алгоритмическим задачам: бывают ситуации, в которых *жадный* (greedy) алгоритм, стремящийся максимизировать выгоду (в том или ином смысле) каждого отдельного шага, приводит к оптимальному решению. Таким случаям и посвящена эта глава. Наш первый пример — минимальные покрывающие (остовные) деревья.

5.1. Покрывающие деревья

Мы хотим связать несколько компьютеров в сеть, соединяя их попарно. Это соответствует задаче на графе, где вершины — компьютеры, (неориентированные) рёбра — возможные связи, и надо получить связный граф. Для каждого соединения известна стоимость обслуживания (неотрицательный вес соответствующего ребра). Как найти самый дешёвый вариант?



Ясно, что оптимальный набор рёбер не может содержать циклов, потому что удаление ребра из цикла снижает стоимость без потери связности:

Свойство 1. *Удаление ребра из цикла сохраняет связность графа.*

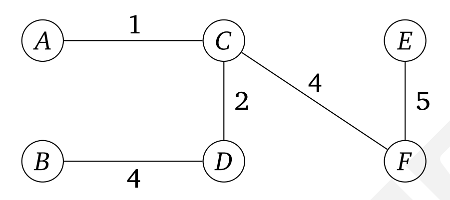
Поэтому решением должен быть связный ациклический граф; неориентированные графы такого вида называются *деревьями*. Необходимое нам дерево с минимальным суммарным весом называется *минимальным покрывающим деревом* (minimum spanning tree, MST). Формально говоря, наша задача такова:

Вход: неориентированный граф $G = (V, E)$, неотрицательные веса рёбер w_e.

Выход: дерево $T=(V,E')$, где $E'\subseteq E$, минимизирующее

$$\mathtt{weight}(T)=\sum_{e\in E'} w_e.$$

В предыдущем примере минимальное покрывающее дерево имеет стоимость 16:



Но это не единственное оптимальное решение. Видите ли вы другое?

Деревья

Дерево — это неориентированный связный ациклический граф.

Свойство 2. *Дерево с n вершинами содержит $n-1$ ребро.*

В самом деле, будем строить дерево, начав с пустого графа и добавляя рёбра по одному. Изначально каждая из n вершин образует компоненту связности. При добавлении ребра две компоненты сливаются (ребро не может соединять две вершины одной компоненты — будет цикл). В итоге остаётся одна компонента, так что понадобится $n-1$ шагов.

Верно и обратное:

Свойство 3. *Всякий связный неориентированный граф $G=(V,E)$, где $|E|=|V|-1$, является деревом.*

Надо показать, что в G нет циклов. Если есть цикл, по свойству 1 можно удалить одно из его рёбер, не нарушая связности (и не меняя числа вершин). Когда циклов не останется, будет дерево, значит, рёбер по-прежнему $|V|-1$, что невозможно, если мы их удаляли.

Иначе говоря, мы можем определить, является ли связный граф деревом, просто сосчитав, сколько в нем рёбер.

Свойство 4. *Неориентированный граф является деревом тогда и только тогда, когда между любыми двумя вершинами существует ровно один путь.*

(Имеются в виду пути, не проходящие по ребру туда-обратно.) Если в дереве есть два пути между одними и теми же вершинами, то в некоторый момент первый путь отклоняется от второго и после этого в какой-то момент снова с ним пересекается. При этом образуется цикл. С другой стороны, если в графе между любыми двумя вершинами имеется путь, то граф связен. Если этот путь единственен, то циклов нет (в цикле между любыми двумя вершинами есть два пути).

5.1.1. Жадная стратегия

Алгоритм Крускала (Kruskal) поиска минимального покрывающего дерева начинает с графа без рёбер и добавляет их; при этом

> на каждом шаге выбирается ребро наименьшего веса, которое не создаёт цикла.

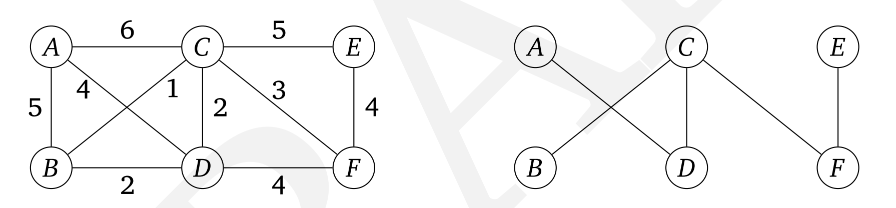
Этот алгоритм *жадный* (можно было бы опасаться, что добавление какого-то лёгкого ребра потом вынудит нас добавить очень тяжёлое, но мы про это не думаем).

На рисунке 5.1 мы упорядочиваем рёбра по возрастанию (неубыванию) весов:

$$B–C,\ C–D,\ B–D,\ C–F,\ D–F,\ E–F,\ A–D,\ A–B,\ C–E,\ A–C.$$

Добавляем их одно за другим, если не возникает цикла: первые два годятся, третье ребро $B–D$ при добавлении создаёт цикл, так что мы его пропускаем, и т. д. Окончательный результат — дерево с минимально возможной стоимостью 14.

Рис. 5.1. Минимальное покрывающее дерево, построенное алгоритмом Крускала.



Корректность алгоритма Крускала следует из *свойства разреза*, которое носит общий характер и используется также в других алгоритмах построения минимальных покрывающих деревьев.

5.1.2. Свойство разреза

Предположим, что мы строим минимальное покрывающее дерево, уже выбрали какие-то рёбра и пока на правильном пути (эти рёбра можно дополнить до минимального покрывающего дерева). Какое ребро нужно добавить дальше? Следующее свойство даёт большу́ю свободу выбора.

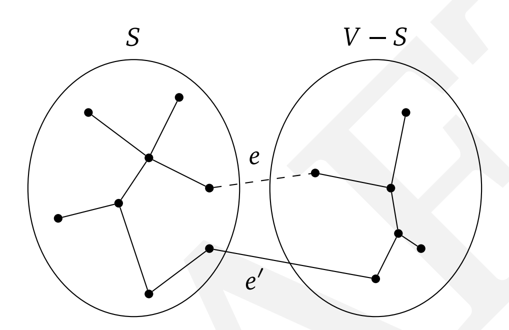
Разрез (cut) — это произвольное разбиение вершин на две части S и $V - S$. Здесь и далее под $V - S$ понимается разность множеств V и S (элементы V, не входящие в S), которая также часто обозначается как $V \setminus S$.

Свойство разреза. *Пусть рёбра множества X входят в минимальное покрывающее дерево для графа $G = (V, E)$. Пусть $S \subseteq V$ — множество вершин, и ни одно ребро из X не соединяет вершину из S с вершиной из $V - S$. Наконец, пусть e — ребро с наименьшим весом, соединяющее S и $V - S$. Тогда $X \cup \{e\}$ является частью некоторого минимального покрывающего дерева.*

Данное свойство утверждает, что если стороны разреза пока не соединены рёбрами, то к X можно добавить минимальное из соединяющих их рёбер, причём X останется частью некоторого минимального покрывающего дерева, если это было так до того.

Почему? Пусть T — минимальное покрывающее дерево, содержащее все рёбра из X. Оно может не содержать e (если содержит, всё доказано). В этом случае концы ребра e соединены некоторым путём в T, который вместе с e

Рис. 5.2. При добавлении ребра e (пунктир) к дереву T (сплошные линии) образуется цикл. В этом цикле есть ещё как минимум одно ребро, пересекающее разрез $(S, V - S)$ (на рисунке e').



образует цикл. Поскольку e пересекает разрез, то в этом цикле есть и другое ребро e', пересекающее разрез (перейдя на другую сторону разреза, мы должны вернуться обратно). Это ребро лежит в дереве T. Замена e' на e в этом дереве даёт связный граф с тем же числом рёбер, то есть тоже дерево. Получили новое покрывающее дерево, которое не хуже старого (вес e' не меньше веса e, который был минимальным по предположению среди всех рёбер, пересекающих разрез) и содержит $X \cup \{e\}$, что и требовалось. (Заметим, что выкинутое ребро e' пересекало разрез и потому не могло быть в X.)

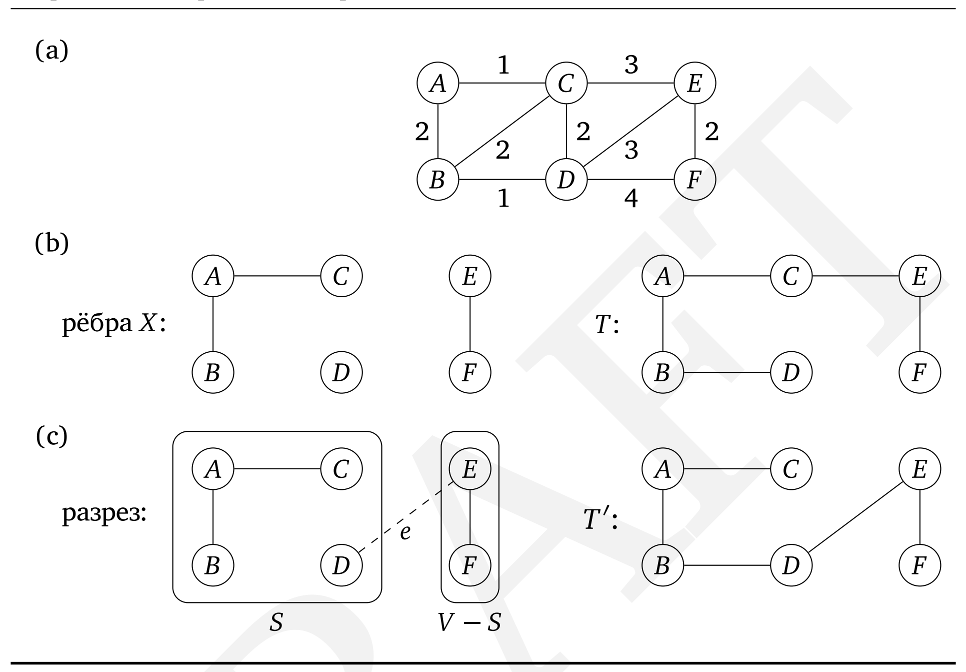
Рис. 5.3 иллюстрирует свойство разреза. Где там ребро e'?

5.1.3. Алгоритм Крускала

Теперь легко показать, что алгоритм Крускала корректен. В каждый момент выбранные рёбра образуют лес (набор деревьев). По индукции покажем, что это лес является частью некоторого минимального покрывающего дерева. Следующее добавляемое ребро e соединяет две компоненты, обозначим их через T_1 и T_2. Так как e — ребро наименьшего веса между разными компонентами, оно будет ребром наименьшего веса между T_1 и $V - T_1$. Остаётся воспользоваться свойством разреза.

Опишем некоторые детали реализации. На каждом шаге алгоритм добавляет новое ребро к уже имеющимся. При этом надо уметь проверять для каждого возможного ребра $\{u, v\}$, лежат ли u и v в разных компонентах. А после добавления соответствующие компоненты необходимо объединить. Какая структура данных поддерживает такие операции?

Рис. 5.3. В неориентированном графе (a) выбрано множество рёбер X (b) и некоторый разрез (c), который не пересекают рёбра из X. Ребро e — одно из минимальных рёбер, пересекающих этот разрез. Не всякое минимальное покрывающее дерево T, содержащее X, содержит e — но в T можно заменить одно из рёбер на e, получив минимальное покрывающее дерево T', содержащее $X \cup \{e\}$.



Для этого используется *система непересекающихся множеств* (disjoint sets); эти множества будут компонентами связности текущего графа. Изначально каждая вершина составляет отдельную компоненту, и мы создаём соответствующие множества с помощью операции

MakeSet(x): создать одноэлементное множество $\{x\}$.

Мы проверяем, лежат ли множества в одной компоненте, сравнивая коды соответствующих компонент; эти коды даёт нам процедура

Find(x): какое множество содержит x?

При добавлении ребра, связывающего вершины x и y, мы объединяем две компоненты с помощью процедуры

Union(x, y): объединить множества, содержащие x и y.

Если нам позволено обращаться к структуре данных, реализующей такие операции, то алгоритм Крускала записать легко (рис. 5.4). Он использует $|V|$ операций MakeSet, $2|E|$ операций Find и $|V| - 1$ операцию Union.

Рис. 5.4. Алгоритм Крускала построения минимального покрывающего дерева.

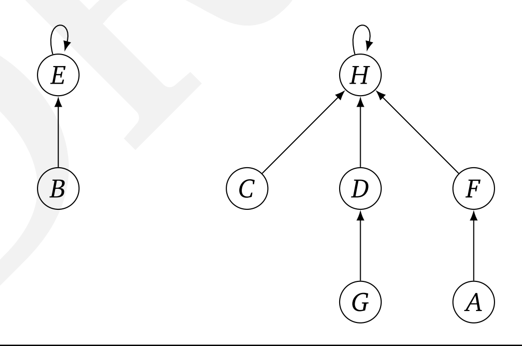
процедура Kruskal(G, w)
{Вход: связный неориентированный граф $G = (V, E)$
  с весами рёбер w_e.}
{Выход: X — минимальное покрывающее дерево (множество рёбер).}
для всех вершин $u \in V$:
  MakeSet(u)
$X \leftarrow \{\}$
упорядочить рёбра множества E по весу
для всех рёбер $\{u, v\} \in E$ в этом порядке:
  если Find$(u) \neq$ Find(v):
    добавить ребро $\{u, v\}$ к X
    Union(u, v)

5.1.4. Непересекающиеся множества: реализация

Объединение по рангу

Один из способов хранения множества — ориентированное дерево (рис. 5.5). Вершины этого дерева — элементы множества. Каждая вершина, кроме корня, хранит также указатель на родителя; корень хранит указатель на самого себя. Корень дерева можно рассматривать как код, или *представитель* (representative), соответствующего множества.

Рис. 5.5. Представление множеств $\{B, E\}$ и $\{A, C, D, F, G, H\}$ в виде ориентированных деревьев.



Помимо родительского указателя π каждая вершина также имеет *ранг*, о котором позже.

процедура MakeSet(x)
$\pi(x) \leftarrow x$
rank$(x) \leftarrow 0$

```
функция FIND(x)
пока x ≠ π(x):
   x ← π(x)
вернуть x
```

Как и следовало ожидать, операция MAKESET занимает время $O(1)$. А операция FIND проходит по родительским указателям к корню, поэтому занимает время, пропорциональное высоте дерева. Строится дерево в ходе операции UNION, которую надо спроектировать так, чтобы по возможности избегать высоких деревьев.

Чтобы объединить два множества (два дерева), можно найти их корни и в одном из них поставить ссылку на другой. Тут есть произвол (какой из корней выбрать). Чтобы высота росла медленнее, логично *подвешивать более низкое дерево к более высокому*. В этом случае высота увеличивается, только если объединяемые деревья имеют одинаковую высоту. Но как узнать высоту? Её можно хранить (и поддерживать при операции соединения) в корне, в поле `rank`. Этот подход называется *объединением по рангу* (union by rank).[1]

```
процедура UNION(x, y)
r_x ← FIND(x)
r_y ← FIND(y)
если r_x = r_y:  ничего делать не надо, выходим
если rank(r_x) > rank(r_y):
   π(r_y) ← r_x {объявляем r_x родителем r_y}
иначе:
   π(r_x) ← r_y {иначе наоборот}
   если rank(r_x) = rank(r_y):   rank(r_x) ← rank(r_y) + 1
```

Пример использования этих операций показан на рис. 5.6.

Последние две строки алгоритма корректируют информацию о высотах поддеревьев, когда это необходимо (когда соединяются деревья одинаковой высоты). По индукции легко проверить, что в поле ранга `rank` всегда будет храниться высота поддерева, растущего из этой вершины. В частности, при подъёме к корневой вершине значения ранга возрастают:

Свойство 1. *Если* $x \neq \pi(x)$, *то* $\mathtt{rank}(x) < \mathtt{rank}(\pi(x))$.

Дерево с рангом k получается только при объединении двух деревьев ранга $k-1$. По индукции отсюда следует (проверьте):

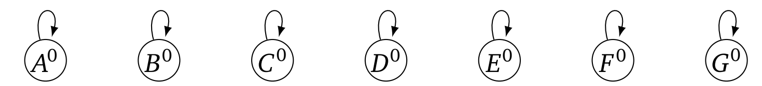
Свойство 2. *Каждая корневая вершина ранга k имеет по меньшей мере 2^k вершин в своём дереве.*

Это верно и для внутренних (не корневых) вершин: вершина ранга k имеет по меньшей мере 2^k потомков (считая её саму). Действительно, при соеди-

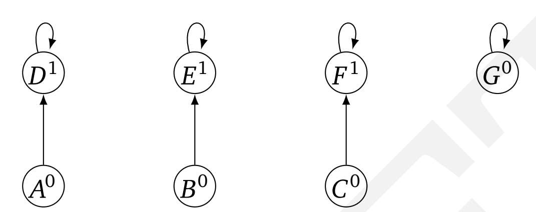
[1]На самом деле нам будет полезно иметь такое поле не только для корней, но и для всех вершин. Видно, что в процедуре UNION в её нынешнем варианте в каждой вершине в поле `rank` остаётся высота поддерева, висящего под ней — которая перестаёт меняться, как только вершина перестала быть корнем. — *Прим. ред.*

Рис. 5.6. Последовательность операций для системы непересекающихся множеств. Ранги обозначены верхними индексами.

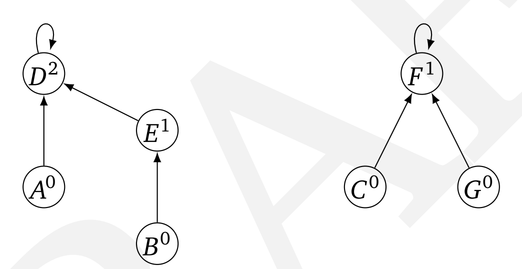
После MAKESET(A), MAKESET(B), ..., MAKESET(G):



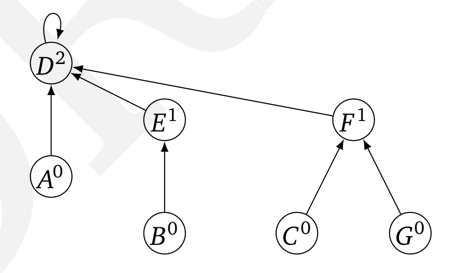
После UNION(A, D), UNION(B, E), UNION(C, F):



После UNION(C, G), UNION(E, A):



После UNION(B, G):



нении двух деревьев у их внутренних вершин не меняются ни ранги, ни потомки, а для корня, становящегося внутренней вершиной, это свойство мы уже знаем. Заметим ещё, что различные вершины ранга k не могут иметь общих потомков, потому что по свойству 1 всякий элемент имеет не более одного предшественника ранга k.

Свойство 3. *Если всего элементов n, то имеется не более $n/2^k$ элементов ранга k.*

В частности, ранг любой вершины не больше $\log_2 n$. Таким образом, все деревья имеют высоту не более $\log_2 n$, и получается оценка сверху на время работы Find и Union.

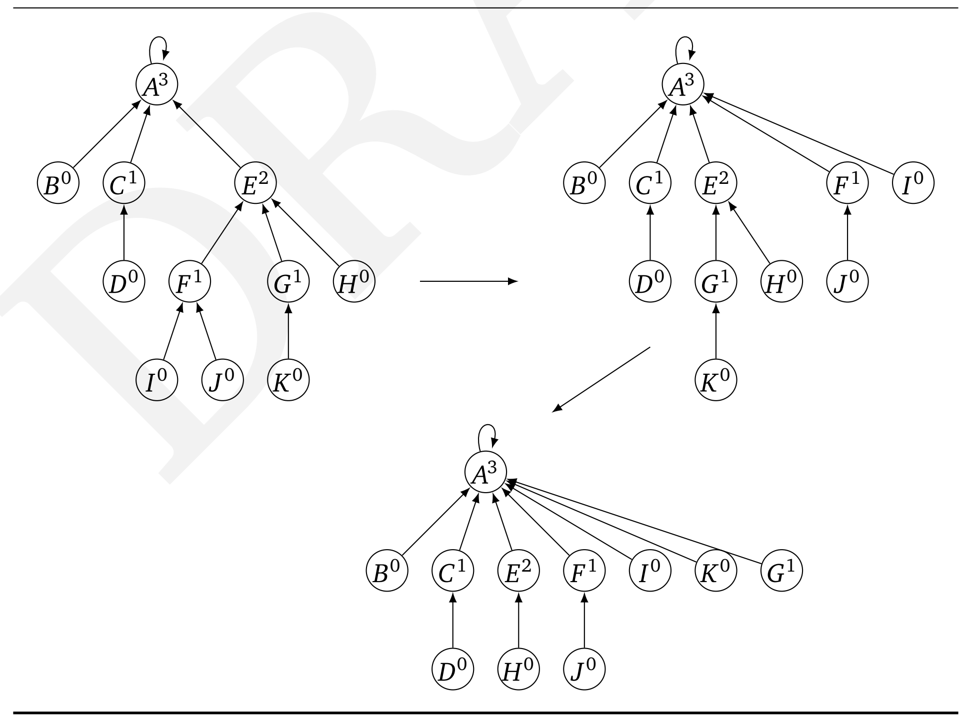
Сжатие путей

Теперь мы можем оценить время работы алгоритма Крускала. Сортировка рёбер займёт время $O(|E| \log |V|)$ (напомним, что $\log |E| = O(\log |V|)$, так как $|E| \leqslant |V|^2$). Столько же уйдёт на объединение и поиск в оставшейся части алгоритма. Казалось бы, нет смысла улучшать нашу структуру данных.

Однако это всё же может иметь смысл. Представим себе, что данные нам рёбра уже упорядочены по весу и на это тратить время не надо. Или, допустим, все веса невелики (например, $O(|E|)$), и тогда сортировку можно сделать за линейное время. В таких случаях «узким местом» становится структура данных, и полезно задуматься об улучшении её производительности, тем более что это совсем несложно, а наша структура данных полезна и в других ситуациях.

В чём состоит улучшение? Как по ходу дела упростить себе будущую работу? В ходе каждого вызова Find мы проходим от вершины к корню — так почему бы нам тут же не прицепить все эти вершины к корню, чтобы не повторять путь в следующий раз? Этот трюк со *сжатием путей* (path compression) показан на рис. 5.7; он лишь немного увеличивает время работы операции

Рис. 5.7. Сжатие путей. Показаны результаты вызовов Find(I) и Find(K).



FIND и легко программируется с помощью рекурсивной процедуры

```
функция FIND(x)
если x ≠ π(x):
   π(x) ← FIND(π(x))
вернуть π(x)
```

Преимущество этой простой модификации проявляется в долгосрочной перспективе, а не мгновенно, поэтому требует особого анализа: мы должны рассмотреть *последовательности* операций FIND и UNION. Эта *амортизированная стоимость* (amortized cost) оказывается немногим больше $O(1)$ и значительно улучшает нашу предыдущую оценку $O(\log n)$.

Почему на каждый запрос в среднем (почти) хватает одной операции? Ведь некоторые операции FIND вполне могут быть долгими? Выполняя такую операцию, мы заодно перевешиваем все проходимые нами вершины сразу под корень, так что второй раз долгой операции с ними не будет. Значит, количество действий, посвящённых данной вершине, будет небольшим. (Тут есть тонкость: вершина, подвешенная к корню, впоследствии может оказаться в глубине дерева, когда этот корень будет соединён с другими корнями, и снова получится долгая операция. Но это, как мы сейчас увидим, бывает редко и на среднее влияет мало.)

Выделим в нашей структуре «верхний уровень», состоящий из корней деревьев, и «нижний уровень», состоящий из внутренних вершин. Существует разделение труда: операции FIND (со сжатием путей или без него) затрагивают только внутренние вершины деревьев, в то время как операции UNION (после того как мы уже нашли корни соединяемых деревьев с помощью FIND) обращаются только к верхнему уровню. Таким образом, сжатие путей никак не влияет на набор корней и на то, какие вершины подчинены данному корню.

Сжатие путей не меняет ранги, но меняет деревья, так что ранг уже не обязан совпадать с высотой поддерева. Заметим, что как только некоторая вершина перестаёт быть корнем, она навсегда остаётся внутренней и её ранг зафиксирован. Таким образом, ранги всех вершин не меняются при сжатии путей, хоть эти числа больше не могут быть интерпретированы как высоты поддеревьев. В частности, свойства 1—3 (с. 133) остаются в силе.[1]

Если имеется n элементов, их ранги лежат в отрезке от 0 до $\log n$ по свойству 3. Разделим ненулевую часть этого интервала на некоторые специально подобранные части (из соображений, которые станут ясны позже):

$$\{1\},\ \{2\},\ \{3,4\},\ \{5,6,\ldots,16\},\ \{17,18,\ldots,2^{16}=65536\},\ \{65537,\ldots,2^{65536}\},\ \ldots$$

Каждая из этих частей имеет вид $\{k+1, k+2, \ldots, 2^k\}$, где k — степень двойки. Количество частей равно $\log^* n$, то есть числу последовательных операций log, которые необходимо применить к n, чтобы получить число, не пре-

[1]Тут надо быть аккуратным. Свойство 2 теперь верно лишь для корневых вершин ранга k (они образуются, когда соединяются две вершины ранга $k-1$). В дальнейшем вершина ранга k может стать внутренней, и при сжатии путей потерять кого-то из своих детей. Но в любом случае эти дети не используются при образовании новых вершин ранга k, так что свойство 3 остаётся в силе. — *Прим. ред.*

Вероятностный алгоритм для задачи о минимальном разрезе

Мы уже говорили о связи между покрывающими деревьями и разрезами. Вот другой пример. Удалим последнее ребро, которое алгоритм Крускала добавляет к покрывающему дереву; дерево разделится на две компоненты и получится разрез $(S, \bar{S})$. Что можно сказать про этот разрез? Предположим, что все веса рёбер одинаковые, а порядок рёбер (в алгоритме Крускала) случайный. Тогда *с вероятностью по крайней мере* $1/|V|^2$ *разрез* $(S, \bar{S})$ *является минимальным* (минимально число рёбер, проходящих между S и $\bar{S}$). Повторив это $O(|V|^2)$ раз (каждый раз выбирая порядок рёбер случайно) и взяв минимальный из найденных разрезов, мы с большой вероятностью (скажем, 99%, или любой другой, если взять подходящую константу в $O(|V|^2)$) получим минимальный разрез в G. Тем самым получается вероятностный алгоритм со временем работы $O(|E||V|^2 \log |V|)$ для невзвешенных минимальных разрезов. Улучшенный вариант этого алгоритма, имеющий время работы $O(|V|^2 \log |V|)$, построил Дэвид Каргер (David Karger), и это самый быстрый из известных сейчас алгоритмов.

Почему разрез последнего шага будет минимальным с вероятностью по меньшей мере $1/|V|^2$? На каждом шаге множество вершин разбито на компоненты связности. Вершины, добавляемые к дереву, соединяют разные компоненты. Количество рёбер, инцидентных каждой компоненте, не меньше C (размера минимального разреза в G), поскольку они соответствуют другому разрезу. Когда осталось k компонент, число пригодных к добавлению рёбер не меньше $kC/2$ (из каждой из k компонент исходит по меньшей мере C рёбер, но надо учесть двойной подсчёт каждого ребра). Можно считать, что на каждом шаге алгоритма мы выбираем случайное ещё не рассмотренное ребро, так что следующее возможное ребро из списка будет входить в минимальный разрез с вероятностью не более $C/(kC/2) = 2/k$. Таким образом, с вероятностью $1 - 2/k = (k-2)/k$ выбор и на следующем шаге не затронет минимального разреза. Поэтому вероятность того, что алгоритм Крускала так и не затронет минимальный разрез вплоть до последнего ребра покрывающего дерева, по меньшей мере

$$\frac{|V|-2}{|V|} \cdot \frac{|V|-3}{|V|-1} \cdot \frac{|V|-4}{|V|-2} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{|V|(|V|-1)}.$$

восходящее единицы. Например, $\log^* 1000 = 4$, так как $\log\log\log\log 1000 \leqslant 1$. На практике будут использоваться только первые пять из указанных интервалов; больше требуется только при $n \geqslant 2^{65536}$, то есть никогда.

Разные вызовы операции FIND занимают разное время. Мы оценим *среднее* время работы при помощи хитрой бухгалтерии. А именно, каждой вершине мы дадим определённое количество денег, так что в сумме окажется роздано не более $n \log^* n$ рублей. Затем мы покажем, что каждый вызов FIND занимает $O(\log^* n)$ шагов плюс некоторое дополнительное время, которое можно «оплатить» деньгами задействованных вершин, по рублю за единицу времени. Таким образом, среднее время для m вызовов FIND рав-

но $O(m \log^* n)$ (непосредственно учтённое) плюс не более $O(n \log^* n)$ за счёт вершин.

Более точно, вершина получает свои деньги, как только она перестаёт быть корнем; в этот момент её ранг фиксируется. Если этот ранг попадает в интервал $\{k+1, \ldots, 2^k\}$, то вершина получает 2^k рублей. По свойству 3 количество вершин ранга больше k не превышает

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \ldots \leqslant \frac{n}{2^k}.$$

Таким образом, общая сумма, выданная вершинам из данного интервала, не превышает n рублей, и так как всего интервалов $\log^* n$, то общая розданная сумма $\leqslant n \log^* n$.

Время выполнения операции Find пропорционально (для простоты считаем — равно) количеству переходов от сына к отцу. При этом ранги возрастают, и в какие-то моменты переходят из группы в группу по нашей классификации. Таких переходов границы не более $\log^* n$ (понимаете, почему это так?). Эти переходы и дают $O(\log^* n)$ шагов, а остальные нужно оплатить за счёт вершин.

При переходе процедуры Find от ребёнка к родителю (если их ранги в одной группе, см. выше) ребёнок платит рубль. При той же операции Find происходит сжатие путей, так что этот ребёнок переусыновляется родителем более высокого ранга. Так может повторяться несколько раз для одного и того же ребёнка — но хватит ли у него денег? Пока ранг родителя будет оставаться в той же группе, что ранг ребёнка, хватит: внутри группы $\{k+1, \ldots, 2^k\}$ будет меньше 2^k шагов. А после этого (когда родители будут из старших групп) за переходы денег у вершин не берут по правилам оплаты.

Это завершает доказательство оценки $O(\log^* n)$ для среднего времени выполнения последовательности операций в расчёте на одну операцию.

5.1.5. Алгоритм Прима

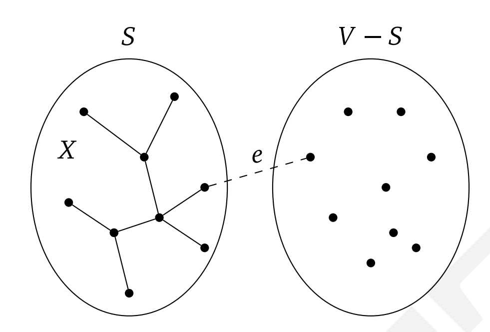
Вернёмся к обсуждению алгоритмов для минимального покрывающего дерева. Свойство разреза показывает, что достаточно следовать такой (жадной) стратегии:

$X \leftarrow \{\}$ {выбранные на данный момент рёбра}
повторять:
  выбрать множество $S \subseteq V$, для которого X не имеет рёбер
    между S и $V - S$
  $e \in E \leftarrow$ ребро минимального веса между S и $V - S$
  $X \leftarrow X \cup \{e\}$
пока $|X| \leqslant |V| - 1$

Популярной альтернативой алгоритму Крускала является алгоритм Прима, в котором X является множеством рёбер некоторого поддерева, а S — множество вершин рёбер из X.

На каждом шаге к дереву добавляется одно ребро, а именно, ребро наименьшего веса среди рёбер, соединяющих S и его дополнение (рисунок 5.8).

Рис. 5.8. Алгоритм Прима: рёбра множества X образуют дерево, а множество S состоит из вершин этого дерева.



Другими словами, на каждом шаге к S добавляется вершина $v \notin S$ минимальной стоимости в смысле такого определения:

$$\texttt{cost}(v) = \min_{u \in S} w(u, v).$$

Это сильно напоминает алгоритм Дейкстры, и на самом деле псевдокод (рисунок 5.9) практически тот же. Единственное различие в том, как задаются приоритеты в очереди. В алгоритме Прима приоритет вершины v определяется значением $\texttt{cost}[v]$, то есть минимальным весом ребра из S в v, в то время как в алгоритме Дейкстры мы сравнивали полные длины путей, а не только их последние рёбра. Но на времени работы это не сказывается (оно зависит от выбранной реализации очереди с приоритетами).

На рисунке 5.9 показана работа алгоритма Прима на маленьком графе с шестью вершинами. Результирующее дерево задаётся массивом указателей на родителя (`prev`).

5.2. Кодирование Хаффмана

Как получаются mp3-файлы?

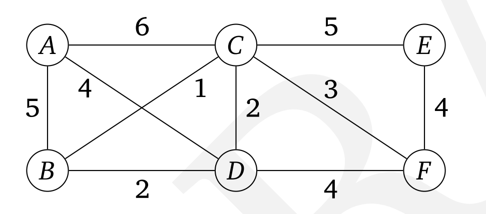
1. Звуковой сигнал (точнее, соответствующий электрический сигнал) измеряется через фиксированные промежутки времени. В идеале эти измерения можно считать последовательностью вещественных чисел $s_1, s_2, \ldots, s_T$. (Например, при частоте 44100 измерений в секунду, как это принято для компакт-дисков, 50-минутная симфония требует $T = 50 \cdot 60 \cdot 44100 \approx 130$ миллионов измерений. Реально большинство записей происходят в режиме «стерео» и два канала дают вдвое больше измерений.)

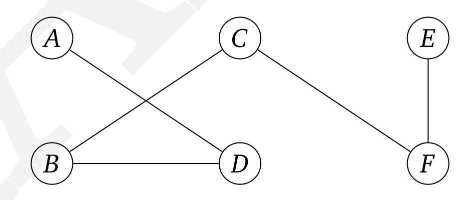
2. Точность измерения ограничена, так что s_t *квантуется* (quantize) и получается число из некоторого конечного множества Γ. (Желательно выбирать это множество аккуратно, чтобы на слух воспроизведённый сигнал был похож на исходный).

3. Полученная последовательность длины T из элементов Γ кодируется битами.

Рис. 5.9. Вверху: алгоритм Прима для построения минимального покрывающего дерева. Внизу: пример работы алгоритма, запущенного из вершины A. Показаны значения массивов `cost`/`prev` и построенное дерево.

```
процедура Prim(G, w)
{Вход: связный неориентированный граф G = (V, E) с весами рёбер w_e.}
{Выход: минимальное покрывающее дерево, заданное массивом prev.}
для всех вершин u ∈ V:
  cost[u] ← ∞
  prev[u] ← nil
выбрать произвольную начальную вершину u_0
cost[u_0] ← 0
H ← MakeQueue(V) {в качестве ключей используются значения cost}
пока H не пусто:
  v ← DeleteMin(H)
  для всех рёбер {v, z} ∈ E:
    если z ∈ H и cost[v] > w(v, z):
      cost[z] ← w(v, z)
      prev[v] ← v
      DecreaseKey(H, z, cost[z])
```





Множество S	A	B	C	D	E	F
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		5/A	6/A	4/A	∞/nil	∞/nil
A, D		2/D	2/D		∞/nil	4/D
A, D, B			1/B		∞/nil	4/D
A, D, B, C					5/C	3/C
A, D, B, C, F					4/F	
A, D, B, C, F, E						

На последнем шаге используется кодирование Хаффмана. Объясним его на простом примере. Пусть, скажем, надо закодировать слово из 130 миллионов букв алфавита $\Gamma = \{A, B, C, D\}$. Можно каждую букву кодировать двумя битами (A как 00, …, D как 11) — понадобится 260 мегабитов. А короче нельзя?

Посмотрим на последовательность более внимательно. Предположим, что мы обнаружили, что разные буквы встречаются с разной частотой:

буква	количество
A	70 миллионов
B	3 миллиона
C	20 миллионов
D	37 миллионов

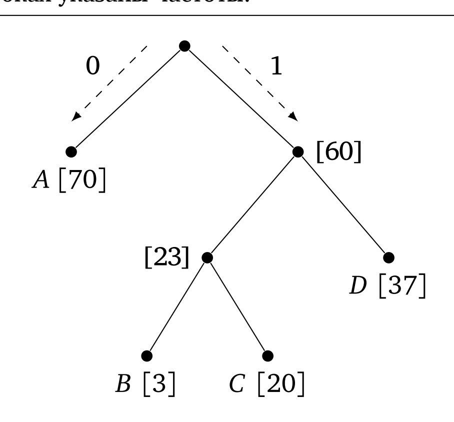
Нельзя ли это использовать и часто встречающиеся буквы кодировать покороче за счёт менее частых? Но если коды букв разной длины, то как провести границы? Скажем, если использовать кодовые слова {0, 01, 11, 001}, то можно раскодировать 001 двумя разными способами. Такого не случится, если код является *беспрефиксным* (prefix-free): никакое кодовое слово не является началом другого.

Кодовые слова (коды букв) беспрефиксного кода можно изобразить вершинами двоичного дерева; эти вершины будут листьями дерева (почему?), и соответствующее кодовое слово можно прочесть на пути из корня в этот лист (поворот налево означает 0, направо — 1). На рисунке 5.10 показан пример такого кода для четырёх символов A, B, C, D. При расшифровке мы читаем последовательность битов, двигаясь по дереву сверху вниз (налево/направо при чтении нуля/единицы). Дойдя до листа, мы говорим себе, что нашли код очередного символа, печатаем этот символ и снова начинаем движение с корня. Тем самым декодирование по-прежнему однозначно, и есть шанс сэкономить: в нашем условном примере (с кодом рис. 5.10) вместо 260 мегабитов получается 213 мегабитов (на 17% лучше).

В этом примере каждая вершина дерева либо является листом, либо имеет двух соседей. Если это не так, то некоторые последовательности (когда мы сворачиваем в сторону, которой в дереве нет) дадут ошибку в расшифровке (при правильном кодировании такого не может получиться). Мы увидим, что в этом случае код — не самый экономный.

Рис. 5.10. Дерево кодов. В квадратных скобках указаны частоты.

символ	код
A	0
B	100
C	101
D	11



Как в общем случае найти оптимальное кодирующее дерево, зная частоты $f_1, f_2, \ldots, f_n$ для кодируемых n символов? Мы хотим минимизировать

$$\text{стоимость дерева} = \sum_{i=1}^{n} f_i \cdot (\text{глубина } i\text{-го символа в дереве})$$

(число битов в кодовом слове равно его глубине в дереве). Под частотами можно понимать количество букв данного вида в кодируемом слове (или долю таких букв, поделив всё на длину слова).

Удобно представлять себе стоимость немного иначе. Хотя нам заданы только частоты листьев, мы можем приписать частоту каждой *внутренней* вершине как сумму частот её листьев-потомков; другими словами, это количество проходов через эту внутреннюю вершину при кодировании и декодировании. Можно представлять себе дело так, что при кодировании мы идём по дереву и в каждой вершине (кроме корня) добавляем бит к коду. Отсюда видно, что минимизируемая нами

> стоимость дерева — это сумма частот всех листьев и внутренних вершин, за исключением корня.

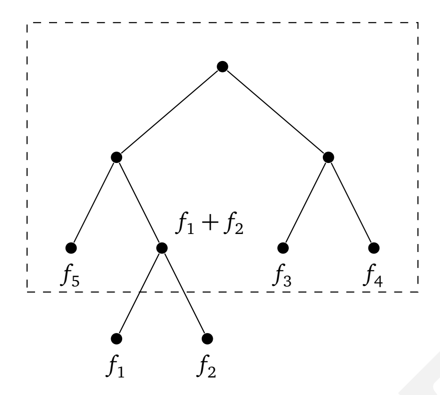
Какими свойствами должен обладать оптимальный код? Посмотрим на самое длинное кодовое слово. Оно должно быть парным, то есть слово, которое отличается в последнем бите (соседний лист), тоже должно быть кодовым словом. Например, если мы какую-то букву кодируем как $x0$, а как $x1$ ничего не кодируем, то почему же мы не используем просто x? Ведь получится тоже беспрефиксный код, и будет короче. (Аналогично можно доказать, что в оптимальном коде любая вершина, не являющаяся листом, имеет двух детей, но это нам не понадобится.)

Итак, посмотрим на два рядом стоящих листа на нижнем уровне дерева. Они будут кодами каких-то двух символов. Если это не два самых редких символа, то можно поменять их местами с этими самыми редкими, и ничего не проиграть. Значит, без ограничения общности можно искать код, в котором *два символа i и j с наименьшими частотами стоят в соседних листьях на нижнем уровне.*

Вспоминая, что стоимость есть число посещений всех листьев, будем считать отдельно заходы в эти два листа (где i и j). Таких заходов будет $f_i + f_j$. А остальные заходы соответствуют ситуации, когда мы объединили символы i и j в один, который встречается $f_i + f_j$ раз, и в качестве кода ему был назначен общий родитель наших листьев. Наоборот, из любого кода для сокращённого алфавита (где i и j склеены) можно сделать код для исходного алфавита, дополнив код объединённого символа нулём и единицей (для i и j).

Отсюда следует, что годится такой (в каком-то смысле жадный) алгоритм: ищем два символа с наименьшими частотами (на рисунке f_1 и f_2), соединяем их в один и сводим задачу к меньшей того же типа.

Другими словами, мы выбрасываем f_1 и f_2 из списка частот, добавляем $(f_1 + f_2)$ и заготавливаем развилку для нижнего уровня. Получается такой ал-



горитм, использующий очередь с приоритетами (с. 110). Он требует времени $O(n \log n)$, если применять двоичную кучу (раздел 4.5.2).

```
процедура HUFFMAN(f)
{Вход: массив частот f[1...n].}
{Выход: дерево кодирования с n листьями.}
{H — очередь с приоритетами с ключами f[·]}
для i от 1 до n:  INSERT(H, i) с ключом f[i]
для k от n+1 до 2n−1: {нужно n−1 внутренних вершин}
  i ← DELETEMIN(H)
  j ← DELETEMIN(H)
  создать вершину с номером k и детьми i, j
  f[k] ← f[i] + f[j]
  INSERT(H, k) с ключом f[k]
```

Посмотрите теперь на наш игрушечный пример: задаёт ли дерево рис. 5.10 оптимальный код или бывает ещё лучше?

5.3. Формулы Хорна

Мечты об искусственном интеллекте наводят на мысль, что полезно обучить компьютер обращению с логическими формулами. В этом разделе мы рассмотрим формулы из ограниченного класса, называемые формулами Хорна (Horn formulas).

Формулы строятся из *булевых переменных* (Boolean variable), которые могут принимать значения `true` (истина) или `false` (ложь). Переменные соответствуют каким-то утверждениям, типа «убийство произошло на кухне», «дворецкий невиновен» или «полковник спал в 8 вечера».

В формулы будут входить переменные и их отрицания; и те, и другие называют *литералами* (literal). Отрицание $\bar{x}$ («не x») переменной x считается истинным, когда переменная x ложна. Формулы Хорна представляют собой конъюнкцию утверждений двух видов (о значениях переменных):

1. *Импликации*, левая часть которых — это конъюнкция (логическое И) произвольного количества переменных, а правая часть — одна переменная.

Энтропия

В трёх городах стоит по метеостанции, которая регистрирует дождь, снег или ясную погоду; вот данные по доле таких дней (из общего числа, скажем, в 200 дней):

Явление	А	Б	В
дождь	0,15	0,30	0,20
снег	0,10	0,05	0,30
ясная погода	0,70	0,25	0,30
остальное	0,05	0,40	0,20

Где погода «разнообразнее»? Можно уточнить этот вопрос так: где метеостанции потребуется больше битов, если сжимать информацию о погоде в этом городе оптимальным беспрефиксным кодом? Информация о погоде в городе — это слово длины 200 в четырёхбуквенном алфавите. Алгоритм Хаффмана даёт 290 битов для А, 380 битов для Б и 400 битов для В (где двухбитовый код оказывается оптимальным). В этом смысле погода в А наиболее предсказуемая.

Говоря философски, последовательность событий тем более случайна (непредсказуема), чем хуже она сжимается:

$$\text{более сжимаемое} \equiv \text{менее случайное} \equiv \text{более предсказуемое.}$$

Пусть есть n возможных исходов с вероятностями $p_1, p_2, \ldots, p_n$ (в нашем примере $n = 4$) и последовательность из m испытаний. При большом m число исходов типа i обычно близко к mp_i; предположим для простоты, что равенство точное, а также что все p_i являются степенями двойки (то есть имеют вид $1/2^k$). По индукции можно доказать (упражнение 5.19), что код Хаффмана потребует $\sum_{i=1}^{n} mp_i \log_2(1/p_i)$ битов, то есть

$$H(p_1, \ldots, p_n) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

битов на одно измерение. Эту величину называют *энтропией Шеннона* распределения $p_1, \ldots, p_n$.

Если вероятности не равны в точности степеням двойки, то код Хаффмана требует больше битов, чем говорит формула для энтропии (но можно приблизиться к этой оценке, если использовать более сложные коды, например, кодировать блоки из нескольких букв).

Для честной монеты $(1/2,\ 1/2)$ энтропия равна $\frac{1}{2}\log_2 2 + \frac{1}{2}\log_2 2 = 1$, и тривиальное кодирование исходов нулём и единицей оптимально. Для несимметричной монеты $(3/4,\ 1/4)$ энтропия будет меньше: $\frac{3}{4}\log_2\frac{4}{3} + \frac{1}{4}\log_2 4 = 0{,}81$. Код Хаффмана для двух исходов ничего не экономит, но более сложные коды (скажем, применяемые в стандартных программах сжатия файлов) позволяют приблизиться к этой оценке.

См. также упражнения 5.18 и 5.19.

Они выражают утверждения вида «если все условия слева выполнены, то должно быть верно и условие справа». Например, формула вида

$$(z \wedge w) \Rightarrow u$$

может означать «если полковник спал в 8 вечера и убийство произошло в 8 вечера, то полковник невиновен». Вырожденный (но допустимый) вариант импликации — импликация с пустой левой частью «$\Rightarrow x$», означающая просто, что x верно: «убийство точно произошло на кухне».

2. Полностью *отрицательные дизъюнкты*, состоящие из дизъюнкций (логическое ИЛИ) произвольного количества отрицаний, как в случае

$$(\bar{u} \vee \bar{v} \vee \bar{y})$$

(«все не могут быть невиновными»).

Мы хотим определить по заданной формуле Хорна (то есть набору ограничений этих двух типов), существует ли непротиворечивое объяснение — набор значений `true`/`false` переменных, который удовлетворяет всем ограничениям. Это также называется *выполняющим набором значений* (satisfying assignment).

Два вида ограничений «тянут» значения переменных в противоположные стороны: импликации требуют присвоить некоторым переменным `true`, а отрицательные дизъюнкты говорят, что им нужно присвоить `false`. Наш алгоритм будет таким: мы начинаем с присвоения значения `false` всем переменным. Затем мы присваиваем некоторым из них `true`, но очень неохотно (только если какая-то импликация нас к этому вынуждает). Когда все требования такого рода выполнены, мы переходим к отрицательным дизъюнктам и проверяем, что они все выполнены. Наш алгоритм получается не столько жадным, сколько недоверчивым: он полагает какую-то переменную истинной, лишь если его к этому вынудили.

```
{Вход: формула Хорна.}
{Выход: выполняющий набор, если есть.}
присвоить значение false всем переменным
пока есть не выполненная импликация:
{слева все переменные истинны, а справа переменная ложна}
  присвоить переменной в правой части значение true
если все полностью отрицательные дизъюнкты выполнены:
  вернуть набор значений
иначе:
  вернуть «формула не выполнима»
```

Например, пусть формула имеет вид

$$(w \wedge y \wedge z) \Rightarrow x,\ (x \wedge z) \Rightarrow w,\ x \Rightarrow y,\ \Rightarrow x,\ (x \wedge y) \Rightarrow w,\ (\bar{w} \vee \bar{x} \vee \bar{y}),\ (\bar{z}).$$

Мы начинаем со всех значений `false` и далее видим, что x должно быть `true` в силу импликации «$\Rightarrow x$». Потом мы замечаем, что y также должно быть `true`, так как $x \Rightarrow y$, и так далее.

Почему этот алгоритм корректен? Если он возвращает набор значений, то все условия выполнены по построению. Так что надо лишь убедиться в том, что если алгоритм не находит выполняющего набора, то его действительно не существует. Это так, потому что благодаря недоверчивости алгоритма сохраняется следующий инвариант:

> если для некоторого множества переменных установлено `true`, то они должны иметь значение `true` в *любом* выполняющем наборе значений.

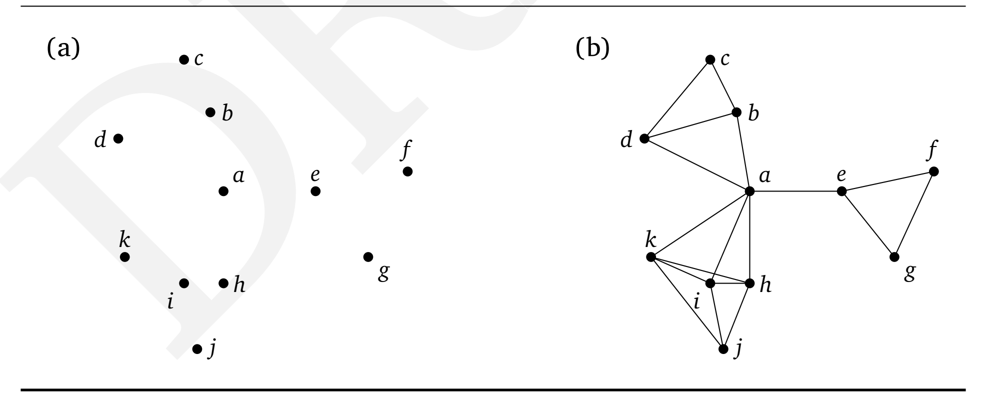
Таким образом, если значения переменных после выхода из цикла не удовлетворяют какой-то отрицательной дизъюнкции, то выполняющего набора значений вообще не существует.

Формулы Хорна лежат в основе языка логического программирования Пролог; интерпретаторы Пролога работают на основе нашего жадного алгоритма. Кстати, этот алгоритм можно реализовать со временем, линейно зависящим от длины формулы. Ясно ли вам, как именно (упражнение 5.33)?

5.4. Покрытие множествами

Есть 11 деревень (рис. 5.11) с какими-то расстояниями между ними; в некоторых из них надо открыть (ну, или не закрыть при «оптимизации») школы; требуется, чтобы для каждой деревни была школа на расстоянии не больше 30 км. Какое минимальное число школ нужно?

Рис. 5.11. (a) Одиннадцать деревень. (b) Пары деревень, находящиеся на расстоянии не более 30 километров.



Удобно изобразить деревни вершинами графа, а рёбрами соединить те из них, между которыми расстояние не больше 30 км. Для каждой вершины x рассмотрим множество S_x из самой этой вершины и всех её соседей (деревень на расстоянии не больше 30 км). Вопрос в том, сколько множеств S_x нужно выбрать, чтобы покрыть все деревни.

Таким образом, мы свели дело к общей задаче о *покрытии множествами*:

Вход: множество B и его подмножества $S_1, \ldots, S_m \subseteq B$.
Выход: несколько множеств S_i, которые вместе покрывают всё B.
Стоимость: количество выбранных множеств.

Очевидное жадное решение:

```
пока все элементы B не покрыты:
  взять множество S_i с максимальным числом непокрытых элементов
```

Алгоритм понятный, хотя не оптимальный: в нашем примере он поместит школу в a (наибольшее число соседей). Останутся непокрытыми деревни c, j и f, и никакие две их них нельзя покрыть сразу, так что всего получается четыре. Однако существует решение только с тремя школами в b, e и i!

К счастью, этот алгоритм всегда даёт решение, не очень далёкое от оптимального:

Утверждение. *Предположим, что B содержит n элементов и оптимальное покрытие состоит из k множеств. Тогда наш жадный алгоритм использует не более $k \ln n$ множеств.*

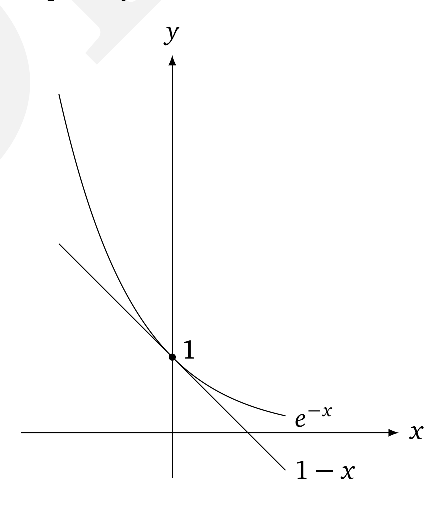
В самом деле, обозначим через n_t число элементов, которые не покрыты после t шагов нашего алгоритма (так что $n_0 = n$). Оставшиеся элементы покрыты k оптимальными множествами. Значит, должно найтись множество, покрывающее хотя бы n_t/k из оставшихся элементов. Следовательно,

$$n_{t+1} \leqslant n_t - \frac{n_t}{k} = n_t\left(1 - \frac{1}{k}\right),$$

что при последовательном применении даёт $n_t \leqslant n_0(1 - 1/k)^t$. Знатоки математического анализа помнят неравенство

$$1 - x \leqslant e^{-x} \quad \text{для всех } x,$$

в котором равенство достигается лишь при $x = 0$ (а не-знатоки могут поверить в него, глядя на картинку).



Получаем теперь

$$n_t \leqslant n_0\left(1-\frac{1}{k}\right)^t < n_0(e^{-1/k})^t = ne^{-t/k};$$

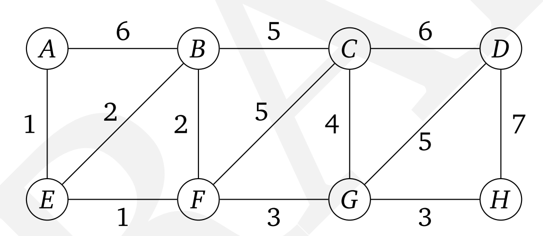
при $t = k \ln n$ число n_t строго меньше $ne^{-\ln n} = 1$, а это означает, что больше никакие элементы покрывать не нужно.

Таким образом, показатель неэффективности (*ошибка приближения*) нашего алгоритма, то есть отношение стоимости найденного решения к оптимальному, не может превышать $\ln n$. Это реальная ситуация: существуют входы, на которых отношение очень близко к $\ln n$ (упражнение 5.34).

Если этот простой алгоритм плох, нельзя ли найти лучший алгоритм (пусть более сложный, но с меньшей ошибкой)? На первый взгляд ничего невероятного в этом нет, но оказывается, что в некоторых предположениях из теории сложности (которые обычно принимают на веру — это станет понятнее, когда мы дойдём до главы 8) можно доказать, что полиномиального алгоритма с меньшей ошибкой приближения не существует.

Упражнения

5.1. Рассмотрим следующий граф.

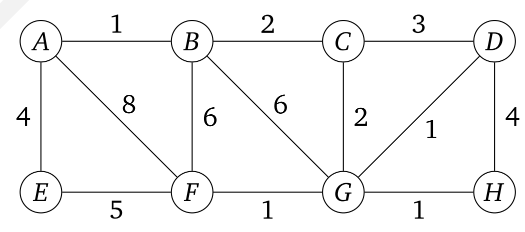


(a) Какова стоимость минимального покрывающего дерева?

(b) Сколько различных минимальных покрывающих деревьев у этого графа?

(c) Примените алгоритм Краскала. В каком порядке добавляются рёбра? Для каждого ребра укажите разрез, обосновывающий его добавление.

5.2. Пусть нам нужно найти минимальное покрывающее дерево для следующего графа.



(a) Примените алгоритм Прима к данному графу. Когда можно выбрать несколько вершин, используйте первую по алфавиту (в частности, запустите

алгоритм из вершины A). Заполните таблицу, показывающую промежуточные значения массива `cost`.

(b) Примените алгоритм Крускала. Приведите состояние системы непересекающихся множеств (то есть леса деревьев) для каждого шага, учитывая сжатие путей.

5.3. Постройте линейный по времени алгоритм для следующей задачи.

Вход: связный неориентированный граф G.
Вопрос: есть ли в G ребро, после удаления которого граф перестаёт быть связным?

Можете ли вы уменьшить время работы алгоритма до $O(|V|)$?

5.4. Покажите, что если неориентированный граф на n вершинах имеет k компонент связности, то в нём хотя бы $n-k$ рёбер.

5.5. Рассмотрим неориентированный граф $G=(V, E)$ с неотрицательными весами на рёбрах $w_e \geqslant 0$. Допустим, мы уже нашли минимальное покрывающее дерево для G, а также кратчайшие пути до всех вершин из некоторой выделенной вершины $s \in V$.

Пусть теперь веса всех рёбер увеличились на единицу (то есть новый вес ребра e есть $w'_e = w_e + 1$).

(a) Может ли измениться минимальное покрывающее дерево? (Докажите, что не может, или приведите пример, когда меняется.)

(b) Могут ли измениться кратчайшие пути? (Докажите, что не могут, или приведите пример, когда меняются.)

5.6. Докажите, что если веса всех рёбер неориентированного графа различны, то минимальное покрывающее дерево единственно.

5.7. Как найти *максимальное* покрывающее дерево графа, то есть покрывающее дерево максимального веса?

5.8. Дан граф $G=(V, E)$ с выделенной вершиной s, веса всех рёбер которого положительны и различны. Может ли быть так, что дерево кратчайших путей из s и минимальное покрывающее дерево графа G не содержат ни одного общего ребра? (Приведите пример или докажите невозможность.)

5.9. Дан неориентированный граф $G=(V, E)$, веса рёбер которого не обязательно различны. Для каждого из утверждений ниже выясните, верно ли оно (приведя доказательство или контрпример).

(a) Если в G больше чем $|V|-1$ ребро и самое тяжёлое ребро уникально, то это ребро не может быть частью минимального покрывающего дерева.

(b) Если в G есть цикл с уникальным самым тяжёлым ребром e, то e не может быть частью минимального покрывающего дерева.

(c) Пусть e — самое лёгкое ребро G. Тогда e входит в некоторое минимальное покрывающее дерево.

(d) Если самое лёгкое ребро G уникально, то оно входит в любое минимальное покрывающее дерево.

(e) Если ребро e является частью некоторого минимального покрывающего дерева графа G, тогда оно является самым лёгким ребром, пересекающим некоторый разрез.

(f) Если в G есть цикл с уникальным самым лёгким ребром e, то e входит в любое MST.

(g) Дерево кратчайших путей, которое выдаёт алгоритм Дейкстры, является минимальным покрывающим деревом.

(h) Кратчайший путь между двумя вершинами является частью некоторого минимального покрывающего дерева.

(i) Алгоритм Прима корректен даже при наличии в графе рёбер отрицательного веса.

(j) Для $r > 0$ определим *r-путь* как путь, все рёбра которого имеют вес меньше r. Если в G есть r-путь из s в t, то любое минимальное покрывающее дерево графа G должно содержать r-путь из s в t.

5.10. Пусть T — минимальное покрывающее дерево графа G, а H — связный подграф G. Покажите, что $T \cap H$ является частью некоторого минимального покрывающего дерева графа H.

5.11. Проследите работу рассмотренной нами реализации системы непересекающихся множеств, начиная с $\{1\}, \ldots, \{8\}$, для операций Union$(1, 2)$, Union$(3, 4)$, Union$(5, 6)$, Union$(7, 8)$, Union$(1, 4)$, Union$(6, 7)$, Union$(4, 5)$, Find(1). Используйте сжатие путей. При объединении множеств с равными рангами представителей подвешивайте вершину с меньшим номером к вершине с бо́льшим номером.

5.12. Пусть в реализации системы непересекающихся множеств используется эвристика объединения по рангу, но не используется эвристика сжатия путей. Приведите последовательность из m операций Union и Find над множеством из n элементов, которая потребует времени $\Omega(m \log n)$.

5.13. Рассмотрим строку над алфавитом из четырёх символов A, C, G, T с частотами 31%, 20%, 9% и 40% соответственно. Каким будет код Хаффмана для этой строки?

5.14. Пусть частоты вхождения символов a, b, c, d, e в строку равны $1/2$, $1/4$, $1/8$, $1/16$, $1/16$ соответственно.

(a) Каким будет код Хаффмана?

(b) Если использовать данный код для сжатия строки длины 1 000 000 с указанными выше частотами вхождения пяти символов, какова будет длина закодированной строки (в битах)?

5.15. Может ли код Хаффмана для трёхбуквенного алфавита оказаться таким (для каких-то частот):

(a) $\{0, 10, 11\}$;

(b) $\{0, 1, 00\}$;

(c) $\{10, 01, 00\}$?

5.16. Докажите следующие два утверждения о кодах Хаффмана.

(a) Если частота одного из символов превосходит $2/5$, то обязательно найдётся символ, код которого будет состоять из одного бита.

(b) Если частоты всех символов меньше $1/3$, то коды всех символов будут длиннее одного бита.

5.17. Какова максимальная длина кодового слова в коде Хаффмана для n символов? Приведите пример частот, при которых достигается такая длина.

5.18. В таблице ниже приведены частоты букв латинского алфавита (и пробела) в некотором тексте.

пробел	17,3%	r	4,8%	y	1,6%
e	10,2%	d	3,5%	p	1,6%
t	7,7%	l	3,4%	b	1,3%
a	6,8%	c	2,6%	v	0,9%
o	5,9%	u	2,4%	k	0,6%
i	5,8%	m	2,1%	j	0,2%
n	5,5%	w	1,9%	x	0,2%
s	5,1%	f	1,8%	q	0,1%
h	4,9%	g	1,7%	z	0,1%

(a) Постройте код Хаффмана для данного алфавита и частот.

(b) Сколько битов в среднем требует этот код в расчёте на букву (в этом тексте)?

(c) Найдите энтропию (см. врезку на с. 144)

$$H = \sum_{i=0}^{26} p_i \log_2 \frac{1}{p_i}.$$

Больше или меньше это число, чем средняя длина, вычисленная в предыдущем пункте?

(d) Как вы думаете, насколько код Хаффмана близок к оптимальному для английских текстов? Какие свойства английского языка, помимо частот букв, можно было бы использовать для сжатия?

5.19. *Энтропия.* Обозначим через $p_1, \ldots, p_n$ вероятности n исходов случайного эксперимента.

(a) В этом пункте будем считать, что все p_i являются отрицательными степенями двойки (вида $1/2^k$). Рассмотрим выборку из m исходов из нашего распределения, в которой исход i встречается ровно mp_i раз. Докажите, что при кодировании методом Хаффмана данной последовательности получится строка длины

$$\sum_{i=1}^{n} mp_i \log_2 \frac{1}{p_i}.$$

(b) Рассмотрим теперь произвольное распределение (вероятности не обязательно являются обратными степенями двойки). Энтропия

$$\sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

часто рассматривается как *количество информации* в исходе эксперимента с таким распределением. Для каких распределений (при данном n) энтропия будет максимальной? А минимальной?

5.20. Постройте алгоритм, который получает на вход дерево и за линейное время определяет, есть ли в нём *совершенное паросочетание* (perfect matching), то есть множество рёбер, которые покрывают каждую вершину ровно по одному разу.

5.21. Постройте эффективный алгоритм для следующей задачи.

Вход: неориентированный граф $G = (V, E)$ с положительными весами w_e на рёбрах.
Выход: множество рёбер с минимальным суммарным весом, удаление которых делает граф ациклическим.

5.22. В данном упражнении мы построим ещё один алгоритм для построения минимальных покрывающих деревьев, основанный на следующей идее.

Рассмотрим цикл в графе, и пусть e — самое тяжёлое ребро этого графа. Тогда найдётся MST, не содержащее e.

(a) Докажите это свойство.

(b) Докажите, что можно искать минимальное покрывающее дерево для неориентированного граф $G = (V, E)$ с весами на рёбрах $\{w_e\}$ следующим образом:

```
упорядочить рёбра по весу
для всех рёбер e ∈ E в порядке уменьшения веса w_e:
  если e входит в некоторый цикл графа G (текущего):
    G ← G − e {удаление ребра e из G}
вернуть G
```

(c) На каждой итерации данный алгоритм должен проверить, есть ли в графе цикл, содержащий заданное ребро e. Как сделать это за линейное время?

(d) Каково время работы полученного алгоритма как функция от $|E|$?

5.23. Дан граф $G = (V, E)$ с положительными весами на рёбрах и его минимальное покрывающее дерево $T = (V, E')$ (будем считать, что G и T заданы списками смежности). Изменим теперь вес какого-то ребра $e \in E$ с $w(e)$ на $\widehat{w}(e)$. Поскольку изменился вес всего одного ребра, мы хотим быстро перестроить имеющееся дерево, а не строить его с нуля. Покажите, как сделать это за линейное время, в каждом из четырёх приведённых ниже случаев.

(a) $e \notin E'$ и $\widehat{w}(e) > w(e)$.

(b) $e \notin E'$ и $\widehat{w}(e) < w(e)$.

(c) $e \in E'$ и $\widehat{w}(e) < w(e)$.

(d) $e \in E'$ и $\widehat{w}(e) > w(e)$.

5.24. В этом упражнении мы ищем покрывающее дерево небольшого веса с дополнительными свойствами.

Вход: неориентированный граф $G = (V, E)$, веса рёбер w_e, подмножество вершин $U \subset V$.
Выход: самоё лёгкое покрывающее дерево, в котором все вершины множества U являются листьями (могут быть и другие листья).

(Искомое дерево может не быть минимальным покрывающим деревом.)

Постройте алгоритм со временем работы $O(|E| \log |V|)$ для этой задачи. (Подсказка: что останется, если выкинуть из оптимального решения вершины множества U?)

5.25. Двоичный счётчик неограниченной длины (реализованный как массив битов) поддерживает две операции: Increment (увеличение счётчика на единицу) и Reset (сброс в ноль). Докажите, что суммарное время работы любой последовательности из n операций Increment и Reset, применённых к изначально нулевому счётчику, есть $O(n)$. Другими словами, амортизированное время работы одной операции есть $O(1)$.

5.26. При автоматическом анализе программ возникает такая задача. Для заданного множества переменных $x_1, \ldots, x_n$ имеется набор ограничений-равенств вида «$x_i = x_j$» и ограничений-неравенств вида «$x_i \neq x_j$». Нужно определить, можно ли выполнить их все (считая, что доступны как минимум n различных значений для этих переменных). Например, система из ограничений

$$x_1 = x_2, \quad x_2 = x_3, \quad x_3 = x_4, \quad x_1 \neq x_4$$

несовместна. Постройте эффективный алгоритм, который проверяет совместность заданной системы из m ограничений от n переменных.

5.27. *Графы с заданными степенями вершин.* Дана последовательность n положительных целых чисел $d_1, d_2, \ldots, d_n$. Мы хотим научиться эффективно проверять, существует ли неориентированный граф $G = (V, E)$ с точно такими степенями вершин. Нас интересуют графы без петель (рёбер из вершины в себя) и кратных рёбер.

(a) Укажите последовательность d_1, d_2, d_3, d_4, для которой $d_i \leqslant 3$ при всех i и $d_1 + d_2 + d_3 + d_4$ чётно, но подходящего графа не существует.

(b) Пусть $d_1 \geqslant d_2 \geqslant \ldots \geqslant d_n$ и существует граф с вершинами $v_1, \ldots, v_n$, имеющими степени вершин $d_1, \ldots, d_n$. Мы хотим доказать, что можно преобразовать граф в другой с теми же вершинами и степенями, в котором соседями v_1 являются $v_2, v_3, \ldots, v_{d_1+1}$. Это можно делать постепенно.

- Пусть соседи у v_1 другие. Покажите, что найдутся $i < j \leqslant n$ и $u \in V$, для которых $\{v_1, v_i\}, \{u, v_j\} \notin E$ и $\{v_1, v_j\}, \{u, v_i\} \in E$.

- Покажите, как преобразовать G в $G' = (V, E')$ с теми же степенями, где $(v_1, v_i) \in E'$.
- Наконец, покажите, что найдётся граф с теми же степенями, в котором соседями вершины v_1 будут вершины $v_2, v_3, \ldots, v_{d_1+1}$.

(c) Используя только что доказанное утверждение, постройте алгоритм, который по последовательности положительных целых чисел $d_1, \ldots, d_n$ (не обязательно упорядоченной) определяет, существует ли граф с такими степенями вершин. Время работы алгоритма должно быть полиномиальным по n и $m = \sum_{i=1}^{n} d_i$.

5.28. Алиса решает, кого из своих n друзей позвать на вечеринку. Ей хочется, чтобы на вечеринке для каждого участника нашлись бы как минимум пятеро других участников, с которыми он знаком, а также пятеро других участников, с которыми он не знаком.

Придумайте эффективный алгоритм, который получает на вход список всех пар знакомых между собой друзей и определяет максимально возможное количество друзей, которое Алиса может позвать на вечеринку.

5.29. *Беспрефиксным кодом* (prefix-free encoding) для алфавита Γ называется набор кодовых двоичных слов (для всех элементов Γ), в котором ни одно из кодовых слов не является префиксом (началом) другого.

Покажите, что для длин кодовых слов $k_1, \ldots, k_n$ выполняется неравенство $2^{-k_1} + \ldots + 2^{-k_n} \leqslant 1$. (Подсказка: кодовые слова можно разместить в двоичном дереве и посмотреть на размеры поддеревьев.)

5.30. *Троичный код Хаффмана.* Компания «Троичные диски» разработала жёсткие диски, хранящие данные в троичной системе счисления. Каждая ячейка диска может хранить число 0, 1 или 2. Помогите компании приспособить метод Хаффмана к её новой технологии. Именно, алгоритм должен по n частотам $f_1, \ldots, f_n$ построить беспрефиксный код из слов в алфавите $\{0, 1, 2\}$, обеспечивающий максимальное сжатие исходного текста.

5.31. Предположим, что для последовательности символов нам даны не только частоты f_i (для каждого символа i), но и их «стоимости» c_i, так что общая стоимость кодирования, если i-й символ кодируется битовой строкой длины l_i, равна $\sum_i f_i \cdot c_i \cdot l_i$.

Модифицируйте алгоритм Хаффмана для поиска беспрефиксного кода минимальной стоимости.

5.32. К серверу приходят одновременно n клиентов. Для клиента i известно время его обслуживания t_i. Время ожидания клиента определяется как сумма времени обслуживания всех предыдущих клиентов и времени обслуживания его самого. К примеру, если сервер обслуживает клиентов в порядке номеров, то время ожидания клиента i будет равно $\sum_{j=1}^{i} t_j$. Постройте эффективный алгоритм, находящий последовательность обслуживания клиентов с минимальным суммарным временем ожидания клиентов.

5.33. Покажите, что в алгоритме проверки выполнимости формул Хорна (раздел 5.3) время работы можно сделать линейным от длины входной формулы. (Подсказка: используйте ориентированный граф для импликаций, в котором вершинами будут переменные.)

5.34. Докажите, что для любого n, являющегося степенью двойки, найдётся такой вход задачи о покрытии множествами (раздел 5.4), для которого

(i) базовое множество состоит из n элементов;
(ii) оптимальное покрытие состоит всего из двух множеств;
(iii) жадный алгоритм даёт $\log n$ множеств.

Таким образом, оценку $O(\log n)$ для жадного алгоритма нельзя улучшить.

5.35. Докажите, что граф с n вершинами (все рёбра одного веса) имеет не более $n(n-1)$ различных минимальных разрезов.

Глава 6
Динамическое программирование

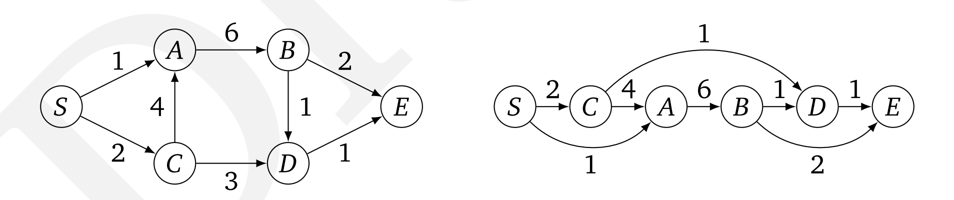
Мы разобрали несколько приёмов построения быстрых алгоритмов («разделяй и властвуй», обход графов, жадные алгоритмы). Если их удаётся применить, есть шансы на простой и быстрый алгоритм. Но так бывает не всегда, и в двух следующих главах мы переходим к «тяжёлой артиллерии» — динамическому и линейному программированию. Хотя они обычно дают не такие быстрые алгоритмы, зато применимы в ситуациях, где ранее разобранные подходы не работают.

6.1. Ещё раз о кратчайших путях в ориентированных ациклических графах

В конце главы 4 мы рассмотрели простой алгоритм нахождения кратчайших путей в ориентированных ациклических графах. Вернёмся к нему, поскольку на его примере хорошо видна идея динамического программирования.

Как мы помним, вершины ориентированного ациклического графа могут быть топологически упорядочены, то есть расположены на прямой так, что все рёбра графа будут идти слева направо (рис. 6.1). Что это даёт? Допу-

Рис. 6.1. Ориентированный граф без циклов и его линеаризация.



стим, что мы хотим найти кратчайшие пути от вершины S до всех остальных вершин. Скажем, в вершину D мы можем добраться только через вершины B или C, поэтому

$$\texttt{dist}[D] = \min\{\texttt{dist}[B] + 1, \texttt{dist}[C] + 3\}.$$

Аналогичные соотношения можно записать для всех вершин графа. Если вычислять значения dist для вершин в порядке их топологической сортировки (слева направо на рис. 6.1), то при обработке вершины v всё, что нужно для вычисления $\texttt{dist}[v]$, уже известно. Таким образом, весь массив dist можно

заполнить за один проход по всем вершинам:

```
присвоить всем dist[·] значение ∞
dist[s] ← 0
для v ∈ V \ {s} в порядке топологической сортировки
  dist[v] ← min{dist[u] + l(u, v): (u, v) ∈ E}
```

Как видно, алгоритм рассматривает *подзадачи* $\{\mathtt{dist[u]} : u \in V\}$ «от простого к сложному»: когда приходит пора решать какую-то подзадачу, для этого всё готово, те задачи, от которых она зависит, уже решены. В рассмотренном нами примере мы брали минимум сумм, вычисленных для каждого из предшественников, но мы могли бы брать и *максимум*, что дало бы нам *максимальные пути*. Или же вместо суммы мы могли бы вычислять произведение, и тогда нашли бы путь с минимальным произведением длин рёбер.

В этом и состоит метод динамического программирования: исходная задача включается в семейство «подзадач» разного размера, и они решаются одна за другой в правильном порядке. Что значит «правильный»? Считаем подзадачи вершинами воображаемого ориентированного ациклического графа (ребро из A в B означает, что для решения B надо сначала решить A); тогда порядок решения подзадач должен быть линеаризацией этого графа.

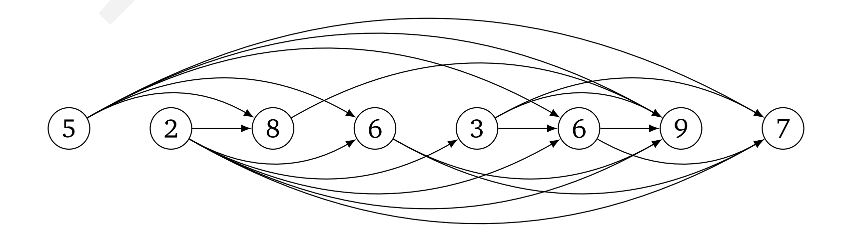
6.2. Наибольшая возрастающая подпоследовательность

В задаче о *наибольшей возрастающей подпоследовательности* (longest increasing subsequence) на вход даётся последовательность чисел $a_1, \ldots, a_n$. Подпоследовательность $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ (где $1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant n$) называется возрастающей, если каждое число в ней строго больше предыдущего. Задача состоит в том, чтобы найти самую длинную такую подпоследовательность. Например, для последовательности 5, 2, 8, 6, 3, 6, 9, 7 ответом будет 2, 3, 6, 9:

5 2 8 6 3 6 9 7

На рисунке соседние элементы оптимальной подпоследовательности соединены рёбрами. Проведём рёбра для всех элементов, которые *могут быть* соседями в возрастающей подпоследовательности: ребро из i в j означает, что $i < j$ и $a_i < a_j$ (рис. 6.2).

Рис. 6.2. Ациклический ориентированный граф возможного соседства.



Получится ациклический граф, в котором возрастающие подпоследовательности соответствуют путям, и надо найти самый длинный путь в графе. Как мы говорили, это можно сделать так:

```
для j от 1 до n
   L[j] ← 1 + max{L[i] : (i, j) ∈ E}
вернуть max_j L[j]
```

Здесь $L[j]$ — максимальная из длин путей, заканчивающихся в j, и, соответственно, из длин возрастающих подпоследовательностей с последним членом a_j (здесь мы называем длиной пути число вершин, а не рёбер, что на 1 больше обычного). Последнее ребро пути должно вести в j из какой-то другой вершины, и $L[j]$ — это максимум из значений $L[\cdot]$ в этих вершинах плюс 1. (Если в j не входит ни одного ребра, то считаем максимум по пустому множеству равным нулю). Так как последней в искомой подпоследовательности может быть любая из вершин, то окончательный ответ — это наибольшее из всех значений $L[j]$.

Как это укладывается в общую схему динамического программирования? Мы выделили множество подзадач $\{L[j] : 1 \leqslant j \leqslant n\}$, при этом *есть порядок на подзадачах и есть рекуррентное соотношение, выражающее решение любой подзадачи через решения предыдущих*. Порядок соответствует росту индексов, а соотношение гласит, что

$$L[j] = 1 + \max\{L[i] : (i, j) \in E\}.$$

Сколько времени требуется для вычисления $L[j]$ таким способом? Вершины i можно найти по списку смежности обращённого графа G^R, который строится за линейное время (упражнение 3.5). Время вычисления $L[j]$ пропорционально входящей степени вершины j, поэтому общее время линейно по $|E|$ и не превосходит $O(n^2)$; худший случай — когда весь массив возрастающий. Получается простое и достаточно эффективное решение.

Надо ещё сказать, как найти саму максимальную возрастающую подпоследовательность, а не её длину. Для этого (как и для кратчайших путей в главе 4), вычисляя $L[j]$, будем в `prev[j]` записывать то i, на котором достигается максимум. Сама оптимальная подпоследовательность легко восстанавливается по таким обратным ссылкам.

6.3. Расстояние редактирования

Часто текстовый редактор, подозревая опечатку, предлагает заменить написанное слово на близкое. Что значит «близкое»? Расстояние между словами можно оценить, записав их одно над другим и добиваясь совпадения в каких-то позициях (как на рисунке для слов SNOWY и SUNNY).

```
S – N O W Y        – S N O W – Y
S U N N – Y        S U N – – N Y
 стоимость: 3        стоимость: 5
```

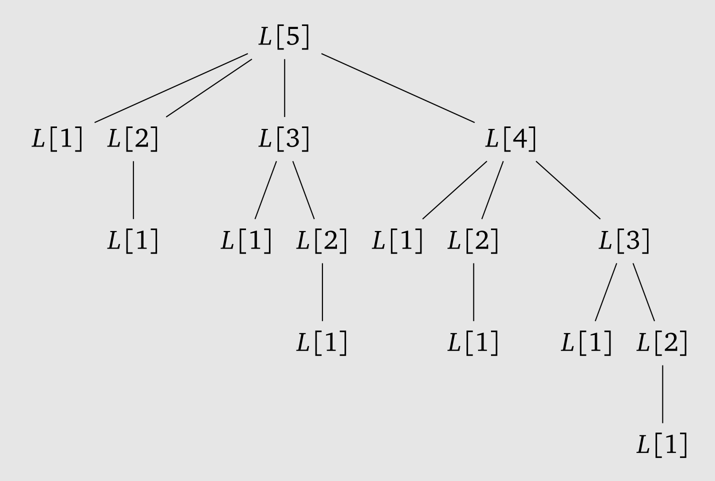
Тире обозначает пробел и может быть добавлено в любые места. *Стоимостью* выравнивания называется количество столбцов, в которых символы различаются, а *расстоянием редактирования* (edit distance) между двумя словами — стоимость их наилучшего возможного выравнивания. (Почему в нашем примере значение 3 минимально?)

Рекурсия не всегда хороша

В задаче о максимальной возрастающей подпоследовательности у нас есть рекуррентная формула для $L[j]$. Почему бы прямо из неё не сделать рекурсивную программу? Дело в том, что время работы будет экспоненциальным. Пусть граф содержит ребра (i, j) для всех $i < j$, то есть последовательность $a_1, a_2, \ldots, a_n$ возрастает. В этом случае

$$L[j] = 1 + \max\{L[1], L[2], \ldots, L[j-1]\}.$$

На рисунке видно, как при вычислении $L[5]$ мы снова и снова решаем одни и те же подзадачи, теряя время.



Для $L[n]$ число вершин в дереве рекурсии будет экспоненциальным (как оценить это число?), и рекурсия надолго зароется в этом дереве.

Почему же рекурсивные алгоритмы были хороши для метода «разделяй и властвуй»? Там задача сводилась к подзадачам, размеры которых были *в разы* меньше. (К примеру, алгоритм сортировки слиянием разбивал массив на два вдвое меньших.) Поэтому дерево рекурсии имело логарифмическую глубину и полиномиальный размер.

А сейчас задача сводится к подзадачам, которые лишь немного меньше: $L[j]$ может зависеть от $L[j-1]$. Дерево рекурсии при этом, как правило, имеет полиномиальную глубину и экспоненциальный размер. Однако *различных* подзадач в дереве не так и много, поскольку они повторяются. Можно быстро найти решение для всех подзадач, решив все различные подзадачи в разумном порядке (и запоминая их ответы).

Название «расстояние редактирования» объясняется тем, что это расстояние можно определить как минимальное число *редакторских правок*, то есть вставок, удалений и замен символов, необходимых для преобразования первой строки во вторую. Например, выравнивание, показанное слева на рисунке, соответствует трём правкам: вставить U, заменить O на N и удалить W. Аналогичное рассуждение показывает, что необходимое число правок не превосходит расстояния редактирования; с другой стороны, каждая правка увеличивает расстояние редактирования не более чем на 1, так что оно не больше числа правок. Определение через число правок хорошо тем, что при этом очевидно *неравенство треугольника*: расстояние A–C не больше суммы расстояний A–B и B–C. С выравниванием это свойство не так очевидно.

Как найти оптимальное выравнивание? Перебирать все варианты долго, но можно воспользоваться динамическим программированием.

Программирование?

Слово «программирование» в названии *динамическое программирование* (как и в «линейном программировании», о котором следующая глава) имело не тот смысл, что сейчас: когда Ричард Беллман придумал этот термин, составление программ не было массовой профессией и названия для этой профессии не было. В те времена программирование означало «планирование» и под «динамическим программированием» понималось оптимальное планирование многоступенчатых процессов. Вершинами графа можно считать состояния процесса, а рёбрами — возможные действия, ведущие к следующему состоянию.

Решение с помощью динамического программирования

Нам надо определить, что мы считаем подзадачами и в каком порядке их решаем. Мы хотим найти расстояние редактирования между $x[1\ldots m]$ и $y[1\ldots n]$. Будем искать расстояние редактирования между *префиксами* первой строки, $x[1\ldots i]$, и *префиксами* второй, $y[1\ldots j]$, обозначив эти расстояния $E[i, j]$ (рис. 6.3). Наша конечная цель — вычисление $E[m, n]$.

Рис. 6.3. Подзадача $E[7, 5]$.

[E X P O N E N] T I A L

[P O L Y N] O M I A L

Чтобы достичь этой цели, нужно как-то выразить $E[i, j]$ через более простые подзадачи. Что известно о наилучшем выравнивании $x[1\ldots i]$ и $y[1\ldots j]$? Ясно, что для самого правого столбца есть три варианта:

$$\begin{array}{ccc} x[i] & - & x[i] \\ - & y[j] & y[j] \end{array}$$

В первом случае стоимость этого столбца равна 1, а оставшиеся столбцы представляют собой выравнивание строк $x[1\ldots i-1]$ и $y[1\ldots j]$, причём оптимальное. А это в точности подзадача $E[i-1, j]$, и она уже решена. Во втором случае стоимость последнего столбца также равна 1, и остаётся задача $E[i, j-1]$. В последнем же случае стоимость равна 1, если $x[i] \neq y[j]$, и 0, если $x[i] = y[j]$, а остаётся задача $E[i-1, j-1]$. Мы не знаем, какой из вариантов реально происходит в оптимальном выравнивании, поэтому проверим все и выберем лучший:

$$E[i, j] = \min\{1 + E[i-1, j], 1 + E[i, j-1], \mathrm{diff}(i, j) + E[i-1, j-1]\},$$

где $\mathrm{diff}(i, j) = 0$, если $x[i] = y[j]$, и 1 в противном случае.

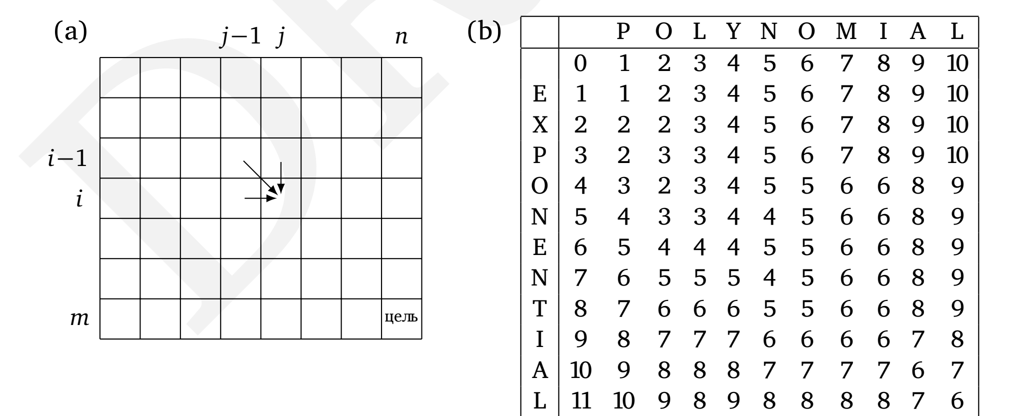
Например, для слов EXPONENTIAL и POLYNOMIAL подзадача $E[4, 3]$ соответствует префиксам EXPO и POL. Правый столбец их наилучшего выравнивания может выглядеть одним из следующих способов:

$$\begin{array}{ccc} \text{O} & - & \text{O} \\ - & \text{L} & \text{L} \end{array}$$

Поэтому $E[4, 3] = \min\{1 + E[3, 3], 1 + E[4, 2], 1 + E[3, 2]\}$.

Решения всех подзадач $E[i, j]$ образуют двумерную таблицу (рис. 6.4). В каком порядке следует решать эти подзадачи? Подойдёт любой порядок, где $E[i-1, j]$, $E[i, j-1]$ и $E[i-1, j-1]$ обрабатываются раньше, чем $E[i, j]$. Например, мы можем заполнять таблицу по строкам, сверху вниз, а внутри строки слева направо. Или, наоборот, по столбцам. В обоих случаях к тому времени, когда мы подойдём к вычислению очередной ячейки, требуемые для неё значения будут уже известны.

Рис. 6.4. (a) Таблица подзадач. Для вычисления значения $E[i, j]$ необходимы значения $E[i-1, j-1]$, $E[i-1, j]$ и $E[i, j-1]$. (b) Заполненная таблица.



		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	6	8	9
N	5	4	3	3	4	4	5	6	6	8	9
E	6	5	4	4	4	5	5	6	6	8	9
N	7	6	5	5	5	4	5	6	6	8	9
T	8	7	6	6	6	5	5	6	6	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Остаётся понять, с чего начинать (когда формула ещё не применима). В данной задаче начальными значениями являются $E[\cdot, 0]$ и $E[0, \cdot]$ и заполняются они очевидным образом: $E[0, j]$ есть расстояние редактирования

между пустой строкой и первыми j символами из y, то есть j. Аналогично $E[i,0]=i$.

Теперь у нас всё готово для записи алгоритма.

```
для i от 0 до m:
  E[i, 0] ← i
для j от 0 до n:
  E[0, j] ← j
для i от 1 до m:
  для j от 1 до n:
    E[i, j] ← min{E[i − 1, j] + 1, E[i, j − 1] + 1, E[i − 1, j − 1] + diff(i, j)}
вернуть E[m, n]
```

Эта процедура заполняет таблицу построчно, слева направо в каждой строке. Обработка каждой ячейки занимает время $O(1)$, поэтому общее время работы пропорционально размеру таблицы, то есть $O(mn)$.

В рассмотренном нами ранее примере расстояние редактирования равно 6:

E	X	P	O	N	E	N	–	T	I	A	L
–	–	P	O	L	Y	N	O	M	I	A	L

Граф задачи и путь в нём

Мы уже говорили, что подзадачи можно рассматривать как вершины графа, в котором рёбра отражают зависимости между ними. В задаче нахождения расстояния редактирования вершинами можно считать клетки (i, j) таблицы, а рёбра соответствуют рекуррентной формуле:

$$(i-1,j)\to(i,j), \qquad (i,j-1)\to(i,j), \qquad (i-1,j-1)\to(i,j)$$

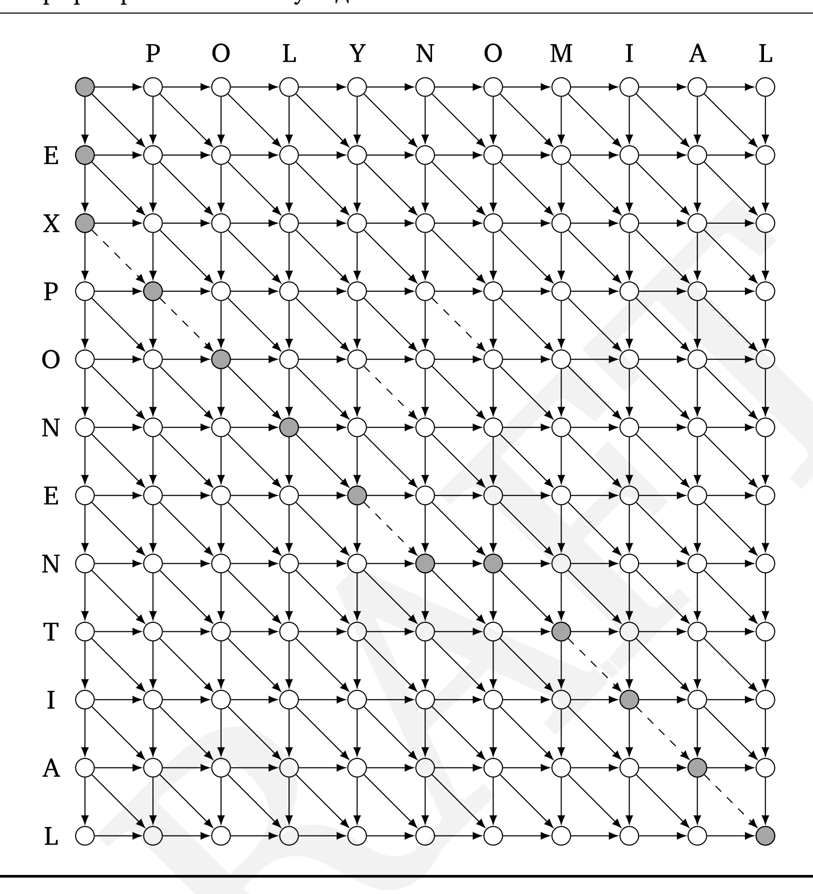
(см. рис. 6.5). Рёбрам этого графа можно так присвоить веса, что оптимальное расстояние редактирования будет равно длине кратчайшего пути. Для этого рёбрам типа $(i-1, j-1)\to(i, j)$ при $x[i]=y[j]$ (пунктирным) присвоим вес 0, а всем остальным — вес 1. Ответом тогда будет просто длина кратчайшего пути из $s=(0,0)$ в $t=(m,n)$. На рисунке показан один из таких путей. Он соответствует как раз оптимальному выравниванию, рассмотренному нами ранее. Каждый шаг вниз на этом пути соответствует удалению, вправо — вставке, а по диагонали — либо совпадению, либо замене.

Теперь видно, что точно так же можно решить и более общий вариант задачи о расстоянии редактирования, когда операции вставки, удаления и замены имеют не обязательно равные стоимости. Нам просто нужно будет изменить веса рёбер в этом графе.

6.4. Задача о рюкзаке

Забравшийся в магазин вор нашёл больше добычи, чем он может унести с собой. Его рюкзак выдерживает не больше W килограммов. Ему надо выбрать

Рис. 6.5. Граф выравнивания и путь длины 6.



какие-то из n товаров веса $w_1, \ldots, w_n$ и стоимости $v_1, \ldots, v_n$. Как найти самый дорогой вариант?[1]

Пусть, например, рюкзак выдерживает $W = 10$ килограммов, а в магазине имеются следующие изделия:

Товар	Вес	Стоимость
1	6	30
2	3	14
3	4	16
4	2	9

Есть две разновидности этой задачи. Если каждый товар имеется в неограниченном количестве, то оптимально будет взять товар номер 1 и две штуки товара номер 4 (общая стоимость: 48). Если же каждый товар есть в

[1]Если этот пример кажется чересчур легкомысленным, несложно выбрать другие математически эквивалентные формулировки.

Часто используемые подзадачи

Включение задачи в семейство подзадач требует изобретательности — но есть несколько стандартных вариантов.

- Вход — последовательность $x_1, x_2, \ldots, x_n$, подзадача — тот же вопрос для префикса $x_1, x_2, \ldots, x_i$ для $i < n$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

  Количество таких подзадач линейно.

- Вход — последовательности $x_1, \ldots, x_m$ и $y_1, \ldots, y_n$, подзадачи — префиксы $x_1, \ldots, x_i$ и $y_1, \ldots, y_j$ для $i < m$ и $j < n$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

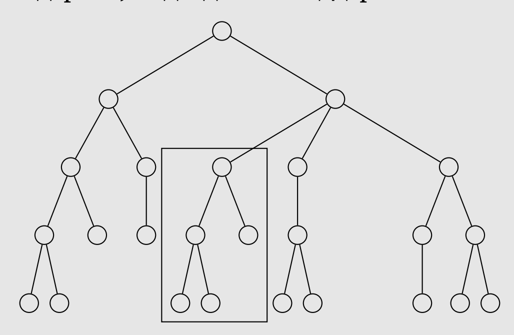
  Количество подзадач есть $O(mn)$.

- Вход — последовательность $x_1, \ldots, x_n$, подзадача — участок $x_i, \ldots, x_j$ для $i < j < n$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

  Количество подзадач есть $O(n^2)$.

- Вход — корневое дерево, подзадача — поддерево.



  Сколько в этом случае будет подзадач?

Первые два случая нам уже встречались, скоро очередь дойдёт и до остальных.

единственном экземпляре (как в художественной галерее), тогда оптимальным будет набор из 1 и 3 (стоимость: 46).

В главе 8 мы увидим, что полиномиальных по времени алгоритмов для этих двух задач, скорее всего, не существует. Однако динамическое програм-

мирование позволяет решить обе задачи (в случае целых весов) за время $O(nW)$, которое вполне допустимо для малых W, но всё же не является полиномиальным, так как размер входа пропорционален $\log W$, а не W.

Задача о рюкзаке с повторениями

Начнём с варианта с повторениями. Как обычно, важно правильно выбрать подзадачи. В данном случае есть два естественных способа: рассмотреть рюкзак меньшей ёмкости $w \leqslant W$ (для краткости максимально допустимый вес мы называем «ёмкостью») или же меньшее число товаров (скажем, товары $1, 2, \ldots, j$, где $j \leqslant n$). Для того чтобы понять, какой подход действительно работает, обычно приходится немного поэкспериментировать.

Попробуем взять рюкзак меньшей ёмкости и положим

$$K[w] = \text{максимально возможная стоимость для рюкзака ёмкости } w.$$

Можем ли мы выразить $K[w]$ через ответы для меньших подзадач? Ясно, что если в оптимальное заполнение рюкзака ёмкости w входит товар i, то без одной штуки этого товара мы получим оптимальное заполнение рюкзака ёмкости $w - w_i$. Другими словами, $K[w]$ — это просто $K[w - w_i] + v_i$ для некоторого i. Мы не знаем, для какого именно i, поэтому нам нужно перебрать все возможные варианты:

$$K[w] = \max_{i\,:\,w_i \leqslant w} \{K[w - w_i] + v_i\},$$

где, как обычно, максимум по пустому множеству считается равным 0. Получаем простой алгоритм:

```
K[0] ← 0
для w от 1 до W:
   K[w] ← max{K[w − w_i] + v_i : w_i ⩽ w}
вернуть K[W]
```

Алгоритм последовательно заполняет массив размера $W + 1$. Значение каждой ячейки вычисляется за время $O(n)$, общее время работы $O(nW)$.

Как обычно, данной задаче соответствует некоторый ориентированный ациклический граф на подзадачах. Решение исходной задачи можно свести к нахождению самого длинного пути в этом графе.

Задача о рюкзаке без повторений

Теперь рассмотрим вариант задачи, когда каждый товар есть в одном экземпляре. Тогда воспользоваться подзадачами из прошлого решения не удаётся, поскольку надо как-то учитывать, что мы уже взяли. Сделаем это, добавив второй параметр $0 \leqslant j \leqslant n$: обозначим через $K[w, j]$ максимальную стоимость унесённого, если разрешается уносить лишь товары $1, \ldots, j$ и общий вес должен быть не больше W. Исходная задача: найти $K[W, n]$.

Теперь нужно научиться выражать $K[w, j]$ через результаты для меньших подзадач. Это несложно. В оптимальном заполнении товар j либо участвует,

О мышах и людях

В каждой живой клетке есть «программа» — последовательность символов алфавита {A, C, G, T} (нуклеотидов), записанная в молекулах ДНК (дезоксирибонуклеиновой кислоты).

У любых двух людей последовательности ДНК (длиной примерно в 3 миллиарда символов) отличаются всего лишь на 0,1%. Но этих трёх миллионов различий достаточно, чтобы люди были совсем разные. Эти различия важны для биологии и медицины — например, анализ ДНК может выявить предрасположенность к некоторым болезням.

ДНК — огромная программа, с которой учёные только начинают разбираться. В ней есть участки с конкретными ролями (производство тех или иных белков), которые называют *генами*. При их обнаружении используются компьютеры, и эта область называется *вычислительной геномикой*. Как могут помочь тут наши алгоритмы?

1. При открытии нового гена в геноме человека один из способов понять его функцию — найти похожие известные гены (возможно, у других хорошо изученных видов — например, мышей). Слово «похожие» можно уточнять с помощью того или иного расстояния.

База данных GenBank, в которой хранятся известные гены, уже имеет общую длину более 10^{10} нуклеотидов и продолжает стремительно расти. Сейчас для поиска обычно используется программа BLAST, при создании которой понадобились и алгоритмические трюки, и биологическая интуиция.

2. При *секвенировании* ДНК (DNA sequencing), то есть определении последовательности символов, обычно разрезают молекулу на куски (подстроки) длиной 500—700 нуклеотидов и все их «читают». Допустим, мы нашли миллиарды таких случайно рассеянных по всей строке фрагментов, но как восстановить исходную последовательность? Ведь ни для одного из этих кусочков мы не знаем, с какой позиции он идёт. Это надо определить, склеивая перекрывающиеся фрагменты, и это была непростая задача. Первые «черновики» полной ДНК человека появились в 2001 году у двух групп: «Human Genome Consortium» (с государственным финансированием) и «Celera Genomics» (частная компания).

3. Когда некий ген найден у нескольких видов, можно ли использовать эту информацию для выяснения происхождения видов?

Мы рассмотрим эти вопросы в упражнениях в конце главы.

либо нет:

$$K[w, j] = \max\{K[w - w_j, j-1] + v_j, K[w, j-1]\}$$

(первый член берётся, только если $w_j \leqslant w$.) Другими словами, можно выразить $K[w, j]$ через результаты подзадач $K[\cdot, j-1]$.

Алгоритм заполняет двумерный массив из $W+1$ строки и $n+1$ столбца. Каждая ячейка заполняется за время $O(1)$. Поэтому несмотря на гораз-

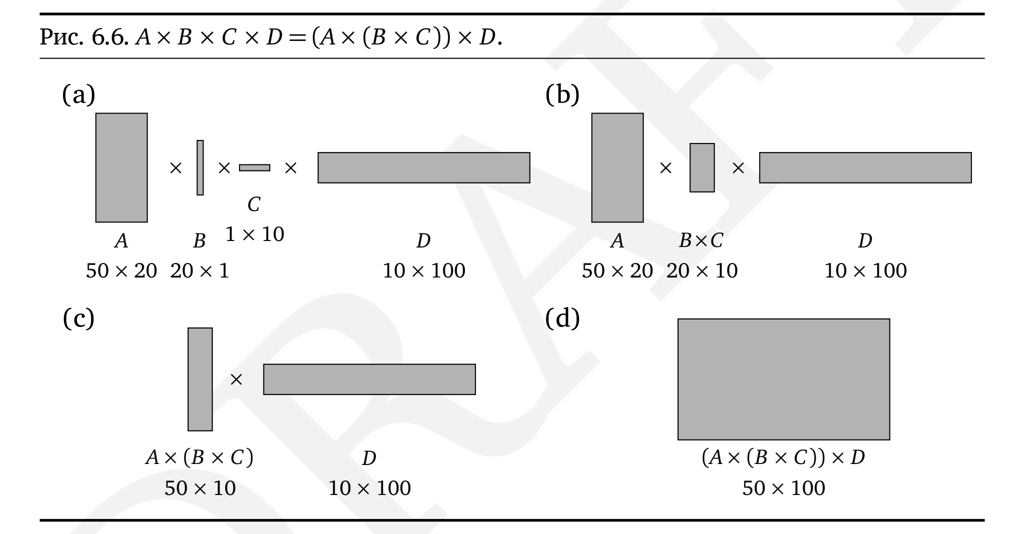
до больший размер таблицы по сравнению с предыдущим алгоритмом общее время работы остаётся прежним: $O(nW)$.

```
для всех j, w:    K[0, j] ← 0;  K[w, 0] ← 0
для j от 1 до n:
  для w от 1 до W:
    если w_j > w:   K[w, j] ← K[w, j − 1]
    иначе:    K[w, j] ← max{K[w, j − 1], K[w − w_j, j − 1] + v_j}
вернуть K[W, n]
```

6.5. Произведение матриц

Пусть нам необходимо вычислить произведение $A \times B \times C \times D$ четырёх матриц размеров 50×20, 20×1, 1×10 и 10×100 (рис. 6.6). Для этого нужно умножать матрицы попарно в каком-то порядке. Как известно, матричное

Рис. 6.6. $A \times B \times C \times D = (A \times (B \times C)) \times D$.



умножение не является *коммутативным* (в общем случае $A \times B \neq B \times A$), но является *ассоциативным* ($A \times (B \times C) = (A \times B) \times C$). Таким образом, мы можем выбирать, в каком порядке перемножать матрицы (или, что то же самое, как расставить скобки). Важен ли тут порядок?

Обычная формула для произведения матриц $m \times n$ и $n \times p$ использует $O(mnp)$ арифметических операций. Считая, что нужно ровно mnp операций, сравним несколько разных способов вычисления $A \times B \times C \times D$:

Порядок	Подсчёт количества операций	Количество операций
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7000

Видно, что порядок умножения сильно влияет на итоговое количество операций. Более того, жадный алгоритм (выбирающий каждый раз две матрицы,

Запоминание

Мы начинали с рекуррентной формулы, выражающей задачи большего размера через меньшие. Она использовалась для заполнения таблицы решений снизу вверх, от подзадач меньшего размера к бóльшим.

Как мы уже говорили, рекурсивный алгоритм, соответствующий рекуррентной формуле, может быть неэффективным, так как одни и те же подзадачи будут решаться снова и снова. Но его можно усовершенствовать, запоминая результаты предыдущих рекурсивных вызовов и используя их при повторных вызовах с тем же входом.

В случае задачи о рюкзаке (с повторениями) такой алгоритм мог бы использовать хеш-таблицу (см. раздел 1.5) для хранения вычисленных ранее значений $K[\cdot]$. При запросе значения $K[w]$ сначала проверялось бы, не присутствует ли оно уже в таблице, и только при его отсутствии производилось бы вычисление. Такой приём называется *запоминанием* (memoization).

```
{В хеш-таблицу помещаются значения K[w] (w — в роли индекса).}
{Изначально таблица пуста.}
функция KNAPSACK(w)
если w содержится в хеш-таблице:
  вернуть K[w]
K[w] ← max{KNAPSACK(w − w_i) + v_i : w_i ⩽ w}
добавить K[w] в хеш-таблицу с ключом w
вернуть K[w]
```

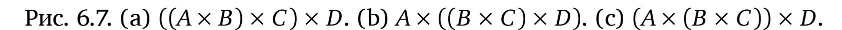
Этот алгоритм никогда не решает дважды одну подзадачу. Его время работы $O(nW)$, как и при использовании динамического программирования без запоминания. Но постоянный множитель, скрытый в $O(\cdot)$, при использовании рекурсии больше, чем в случае динамического программирования (из-за дополнительных проверок и накладных расходов при реализации рекурсии).

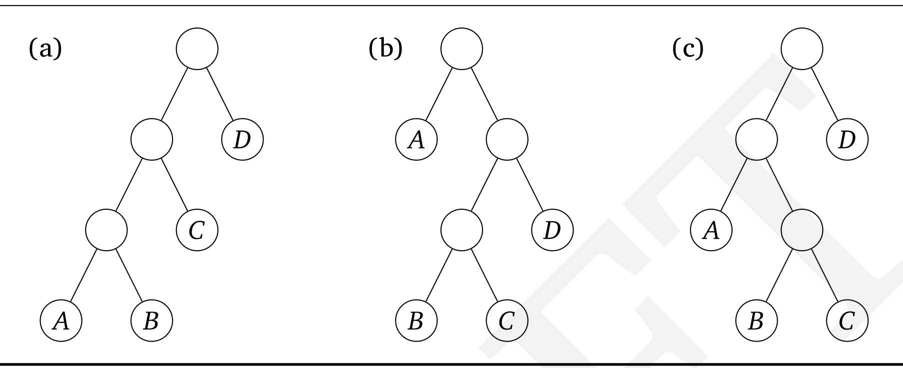
В некоторых случаях, однако, запоминание себя оправдывает. И вот почему: при динамическом программировании решаются все подзадачи, которые потенциально могли бы понадобиться, а при запоминании — только те, которые реально понадобились. Например, пусть W и все веса w_i кратны 100. Тогда подзадача $K[w]$ бесполезна, если w не делится на 100. Рекурсивный алгоритм с запоминанием даже не будет рассматривать эти лишние клетки в таблице.

которые дешевле всего перемножить), приведёт нас ко второму варианту, далёкому от оптимума.

Как же найти оптимальный порядок перемножения матриц $A_1, A_2, \ldots, A_n$ размеров $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$ соответственно? Порядок операций можно изобразить двоичным деревом, листьям которого будут соответствовать исходные матрицы, корню — результат, а внутренним вершинам (у них

два ребёнка) — промежуточные результаты (рис. 6.7). Количество таких деревьев с n листьями зависит от n экспоненциально (упражнение 2.13), поэтому их перебирать долго.


Рис. 6.7. (a) $((A \times B) \times C) \times D$. (b) $A \times ((B \times C) \times D)$. (c) $(A \times (B \times C)) \times D$.



Вспомним, что поддеревья оптимального дерева оптимальны. Какие подзадачи соответствуют поддеревьям? Ими оказываются произведения $A_i \times A_{i+1} \times \ldots \times A_j$. Следуя этой схеме, для $1 \leqslant i \leqslant j \leqslant n$ положим

$$C[i, j] = \text{минимальная стоимость вычисления } A_i \times A_{i+1} \times \ldots \times A_j.$$

Размер подзадачи — это количество матричных операций, то есть $j - i$. При $i = j$ перемножать ничего не надо: $C[i, i] = 0$. При $j > i$ рассмотрим оптимальное поддерево для вычисления $C[i, j]$. В двух его поддеревьях вычисляются $A_i \times \ldots \times A_k$ и $A_{k+1} \times \ldots \times A_j$ для некоторого k между i и j. Стоимость всего поддерева — это сумма стоимостей двух частичных произведений и стоимости их перемножения: $C[i, k] + C[k+1, j] + m_{i-1} \cdot m_k \cdot m_j$. Нам остаётся найти k, для которого эта стоимость окажется минимальной:

$$C[i, j] = \min_{i \leqslant k < j} \{C[i, k] + C[k+1, j] + m_{i-1} \cdot m_k \cdot m_j\}.$$

Получаем такой алгоритм (в котором s — размер подзадачи):

для i от 1 до n: $C(i, i) \leftarrow 0$
для s от 1 до $n - 1$:
  для i от 1 до $n - s$:
    $j \leftarrow i + s$
    $C[i, j] \leftarrow \min\{C[i, k] + C[k+1, j] + m_{i-1} \cdot m_k \cdot m_j : i \leqslant k < j\}$
вернуть $C[1, n]$

Все подзадачи образуют двумерную таблицу. Вычисление значения каждой ячейки требует времени $O(n)$. Поэтому общее время работы алгоритма есть $O(n^3)$.

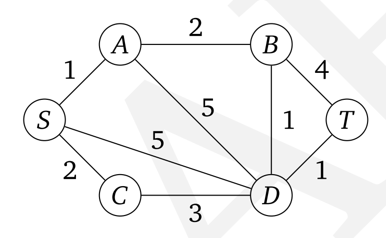
6.6. Кратчайшие пути

Мы начали эту главу с простого алгоритма поиска кратчайшего пути в ориентированном ациклическом графе в качестве примера. Динамическое программирование полезно и в более сложных вариантах поиска кратчайших путей.

Кратчайшие надёжные пути

В практических задачах часто появляются разные дополнительные требования: супердешёвый маршрут с большим числом пересадок вряд ли нужен; пакет байтов, проходящий через большое число маршрутизаторов (даже очень быстрых), может потеряться. В таких ситуациях нужны пути, которые одновременно имеют малую длину (сумму длин рёбер) и малое количество рёбер. Скажем, в графе на рис. 6.8 кратчайший путь из вершины S в T состоит из

Рис. 6.8. Пути из S в T: длина и число рёбер.



четырёх рёбер, в то время как существует немного более длинный путь, проходящий всего по двум рёбрам (и на практике он может быть лучше). Приходим к такой задаче:

Дан взвешенный граф G, вершины s и t и число k. Нужно найти кратчайший путь из s в t, *проходящий не более чем по k рёбрам.*

Можно ли приспособить алгоритм Дейкстры для этой задачи? Сложность в том, что алгоритм Дейкстры сосредоточен на длине кратчайшего пути и не заботится о числе рёбер.

Используя динамическое программирование, мы должны изобрести такие подзадачи, чтобы вся важная информация сохранилась для следующих этапов. В данном случае для каждой вершины v и каждого числа $i \leqslant k$ определим $dist[v, i]$ как *длину кратчайшего пути из s в v, проходящего не более чем по i рёбрам.* Начальные значения $\texttt{dist}[v, 0]$ бесконечны для всех вершин, кроме s, для которой $\texttt{dist}[s, 0] = 0$. Рекуррентная формула очевидна:

$$\texttt{dist}[v, i] = \min_{(u,v)\in E} \{\texttt{dist}[u, i-1] + l(u, v)\}.$$

Кратчайшие пути между всеми парами вершин

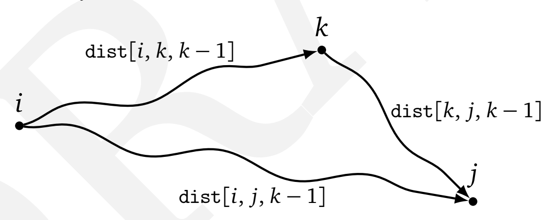
Пусть мы хотим найти кратчайшие пути не только между вершинами s и t, а между *всеми* парами вершин. Можно запустить общий алгоритм нахождения кратчайших путей из пункта 4.6.1 (мы допускаем отрицательные рёбра)

$|V|$ раз, по разу для каждой начальной вершины. Общее время работы будет $O(|V|^2|E|)$. Динамическое программирование даёт более быстрый алгоритм Флойда—Уоршелла, работающий за время $O(|V|^3)$.

Как выбрать подзадачи, чтобы они не были привязаны к начальной или конечной вершинам? Вот идея: будем постепенно разрешать промежуточные вершины («места пересадок»). Если пересадки совсем запрещены, то кратчайший путь из u в v — это просто ориентированное ребро (u, v), если такое ребро есть в графе. Будем разрешать пересадки постепенно, открывая для транзита вершины одну за другой. В конце концов будут разрешены любые пересадки, так что мы найдём длины кратчайших путей между всеми парами вершин графа.

Более конкретно, пронумеруем вершины из множества V от 1 до n и определим $\mathtt{dist}[i, j, k]$ как длину кратчайшего пути из i в j, в котором в качестве промежуточных вершин могут присутствовать только вершины множества $\{1, 2, \ldots, k\}$. Изначально $\mathtt{dist}[i, j, 0]$ — это длина ребра $i \to j$, если такое ребро есть (и ∞, если его нет).

Что происходит, когда к множеству допустимых вершин мы добавляем очередную вершину k? Мы должны проверить для всех пар i, j, даёт ли использование вершины k в качестве промежуточной более короткий путь из i в j. Это несложно: кратчайший путь из i в j, который включает в себя k и другие вершины с меньшими номерами, проходит через k лишь однажды (ведь мы предположили, что в графе нет циклов отрицательного веса). Мы уже знаем длину кратчайшего пути, проходящего через вершины с меньшими номерами, из i в k и из k в j:



Таким образом, добавление k улучшает текущий путь тогда и только тогда, когда

$$\mathtt{dist}[i, k, k-1] + \mathtt{dist}[k, j, k-1] < \mathtt{dist}[i, j, k-1].$$

В этом случае $\mathtt{dist}[i, j, k]$ должно быть соответствующим образом обновлено. Получаем такой алгоритм (Флойда—Уоршелла) с временем работы $O(|V|^3)$:

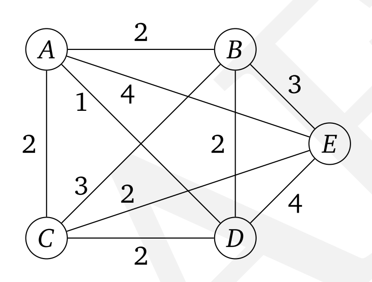
```
для i от 1 до n:
  для j от 1 до n:
    dist[i, j, 0] ← ∞
для всех (i, j) ∈ E:
  dist[i, j, 0] ← l(i, j)
для k от 1 до n:
  для i от 1 до n:
    для j от 1 до n:
      dist[i, j, k] ← min{dist[i, k, k − 1] + dist[k, j, k − 1], dist[i, j, k − 1]}
```

Задача коммивояжёра

Коммивояжёр должен посетить по разу каждый город из большого списка и вернуться в свой родной город. Он знает расстояния между всеми парами городов. В каком порядке лучше всего посещать города, чтобы минимизировать пройденное расстояние?

Пусть города пронумерованы от 1 до n, родной город коммивояжёра имеет номер 1 и пусть $D = (d_{ij})$ — матрица расстояний между городами. Надо найти маршрут, начинающийся и заканчивающийся в городе 1, проходящий все остальные города ровно по одному разу и имеющий минимальную суммарную длину. Даже для графа из пяти городов (рис. 6.9) найти оптимальный маршрут непросто (попробуйте!) — а если городов сотни?

Рис. 6.9. Оптимальный маршрут коммивояжёра имеет длину 10.



Оказывается, эта задача действительно сложная — это одна из классических трудоёмких задач. Хотя постепенно люди придумали много разных способов ускорить её решение, она остаётся сложной. Как мы увидим в главе 8, полиномиальный алгоритм для неё вряд ли существует.

Если перебирать все маршруты (их $(n-1)!$), потребуется время $O(n!)$ (вычисление длины маршрута производится за время $O(n)$). Покажем, что динамическое программирование позволяет решить эту задачу быстрее, хоть и не за полиномиальное время.

Как в данном случае выбрать подзадачи? Поскольку мы ищем маршрут, естественной подзадачей является нахождение начальной части маршрута. Предположим, мы вышли из города 1, посетили несколько городов и сейчас находимся в городе j. Какая информация об этом частичном маршруте существенна для дальнейшего? Надо знать, где мы находимся (j), а также где мы уже побывали (чтобы не идти туда второй раз). Это приводит нас к следующей подзадаче.

> Для подмножества городов $S \subseteq \{1, 2, \ldots, n\}$, включающего 1, и для $j \in S$ обозначим через $C[S, j]$ длину кратчайшего пути из 1 в j, начинающегося в 1 и заканчивающегося в j, проходящего через каждый город из множества S ровно один раз.

При $|S| > 1$ мы полагаем $C[S, 1] = \infty$, поскольку путь не может одновременно начинаться и заканчиваться в городе 1.

Выразим теперь $C[S, j]$ через меньшие. В оптимальном пути из 1 в j предпоследним может быть любой город i из $S - \{j\}$. Без последнего шага мы получим оптимальный путь из 1 в i по $S - \{j\}$, то есть $C[S - \{j\}, i]$. Остаётся добавить длину ребра из i в j, то есть d_{ij}, и перебрать все варианты для i:

$$C[S, j] = \min_{i \in S\,:\, i \neq j} C[S - \{j\}, i] + d_{ij}.$$

Решая подзадачи в порядке возрастания $|S|$, приходим к такому алгоритму:

$C[\{1\}, 1] \leftarrow 0$
для s от 2 до n:
  для всех $S \subseteq \{1, 2, \ldots, n\}$ размера s, содержащих 1:
    $C[S, 1] \leftarrow \infty$
    для всех $j \in S$, $j \neq 1$:
      $C[S, j] \leftarrow \min\{C[S - \{j\}, i] + d_{ij} : i \in S, i \neq j\}$
вернуть $\min_j C[\{1, \ldots, n\}, j] + d_{j1}$

Всего есть не более $2^n \cdot n$ подзадач, и решение для каждой из них находится за линейное время. Поэтому общее время работы есть $O(n^2 2^n)$.

О памяти и времени

Время работы алгоритма, основанного на динамическом программировании, зависит от размера графа подзадач: в большинстве случаев *это просто количество рёбер в графе*. В самом деле, мы просто обходим вершины в порядке топологической сортировки, просматриваем входящие рёбра каждой вершины и, как правило, выполняем $O(1)$ операций для каждого ребра. При этом каждое ребро графа обрабатывается один раз.

Сколько же требуется памяти? Это вопрос более сложный: полная таблица занимает место, пропорциональное количеству вершин (подзадач), но часто можно сэкономить. Дело в том, что результат решения конкретной подзадачи нужно хранить только до тех пор, пока он нужен, то есть пока не будут решены зависящие от него подзадачи большего размера. После этого занимаемая им память может быть использована снова.

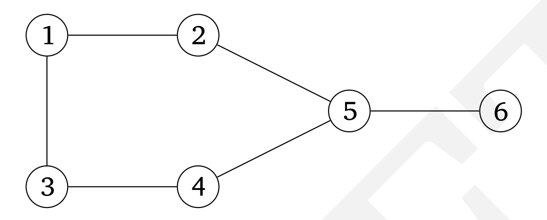
Например, в алгоритме Флойда—Уоршелла значение `dist`$[i, j, k]$ станет ненужным, как только будут вычислены все значения `dist`$[\cdot, \cdot, k+1]$. Поэтому для хранения значений `dist` достаточно иметь всего два массива размером $|V| \times |V|$, один для нечётных значений k, другой — для чётных: при вычислении `dist`$[i, j, k]$ мы затираем значения `dist`$[i, j, k-2]$. (Как и раньше, если нас интересует не только расстояние, но и сам оптимальный путь, нужно об этом позаботиться: заведём массив `prev`$[i, j]$, хранящий предпоследнюю вершину на текущем кратчайшем пути из i в j. Его легко обновлять параллельно с действиями алгоритма.)

Видите ли вы, почему в алгоритме вычисления расстояния редактирования (рис. 6.5) можно обойтись памятью, пропорциональной длине более короткой строки?

6.7. Независимые множества в деревьях

Множество вершин в графе называется *независимым*, если его вершины не соединены рёбрами. Например, на рис. 6.10 множество $\{1, 5\}$ независимо, в отличие от множества $\{1, 4, 5\}$ (вершины 4 и 5 соединены ребром). Самым большим независимым множеством является, например, $\{2, 3, 6\}$.

Рис. 6.10. Размер максимального независимого множества для этого графа равен 3.



Задача нахождения максимального независимого множества в произвольном графе трудна — как и некоторые другие разобранные в этой главе задачи (задача о рюкзаке, задача коммивояжёра). Однако если граф является деревом, то задача может быть решена за линейное время с помощью динамического программирования. Какими же будут соответствующие подзадачи? Как и для задачи умножения матриц, структура дерева (которое мы сделаем корневым) подсказывает выбор подзадач.

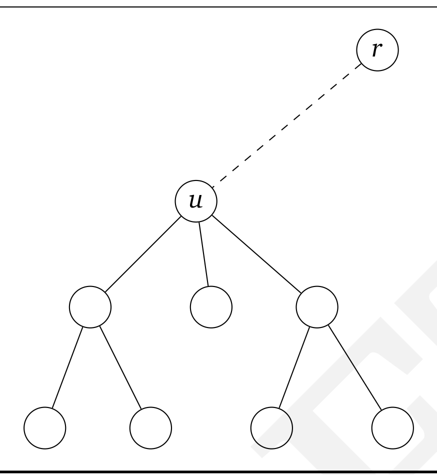
Выберем любую вершину r в качестве корня. Теперь каждая вершина определяет поддерево, подвешенное к ней. Получаем подзадачу $I[u]$: *найти максимальный размер независимого множества в поддереве с корнем в* u. Наша конечная цель заключается в нахождении $I[r]$.

Динамическое программирование, как обычно, решает подзадачи в порядке возрастания размера, от листьев к корню. Предположим, что мы знаем максимальные независимые множества для поддеревьев ниже определённой вершины u. Другими словами, предположим, что мы знаем $I[w]$ для всех w, являющихся потомками u. Как вычислить $I[u]$? Ясно, что либо u входит в максимальное независимое множество, либо не входит (рис. 6.11). Если независимое множество включает u, то не может содержать детей u, поэтому надо сложить ответы для всех «внуков» и учесть u. С другой стороны, если u не входит в независимое множество, то можно сложить максимальные независимые множества для детей. Получаем следующую формулу:

$$I[u] = \max\left\{1 + \sum_{w \text{ — внук } u} I[w],\ \sum_{w \text{ — ребёнок } u} I[w]\right\}.$$

Количество подзадач в точности совпадает с количеством вершин. Аккуратно реализовав алгоритм, можно достичь времени работы $O(|V| + |E|)$.

Рис. 6.11. $I(u)$ равно размеру максимального независимого множества для поддерева с корнем в u. Есть два случая: либо u входит в это множество, либо не входит.



Упражнения

6.1. *Непрерывным участком* называется группа подряд идущих элементов. Например, для последовательности

$$5, 15, -30, 10, -5, 40, 10$$

элементы $15, -30, 10$ образуют непрерывный участок, а $5, 15, 40$ — нет. Постройте линейный алгоритм для следующей задачи:

> *Вход*: список чисел $a_1, a_2, \ldots, a_n$.
> *Выход*: непрерывный участок с максимальной суммой (сумма участка нулевой длины равна нулю).

Для приведённого примера ответом будет $10, -5, 40, 10$ с суммой 55. (Подсказка: для каждого $j \in \{1, 2, \ldots, n\}$ рассмотрите участки, заканчивающиеся в позиции j.)

6.2. Вы едете по длинной дороге, начав у нулевого километрового столба. На дороге у столбов с отметками $a_1 < a_2 < \ldots < a_n$ стоят гостиницы, где можно ночевать. Надо попасть в последнюю (на расстоянии a_n от начала).

Максимально комфортно ехать по 200 км каждый день, но это не всегда возможно (не везде есть гостиницы). Если вы преодолели x километров за день, то *штраф* за этот день составит $(200 - x)^2$. Необходимо спланировать ваше путешествие так, чтобы сумма штрафов за все переезды оказалась минимальной.

Постройте эффективный алгоритм, который находит оптимальный вариант.

6.3. Эффективный менеджер планирует построить несколько новых ресторанов вдоль шоссе. Есть n возможных мест, где можно построить ресторан, на расстояниях $m_1 < m_2 < \ldots < m_n$ от начала шоссе. При этом:

- В каждом из имеющихся n мест можно открыть не более одного ресторана; для каждого места $1 \leqslant i \leqslant n$ известна прибыль $p_i > 0$ от открытого там ресторана.
- Любые два ресторана должны находиться на расстоянии не менее k километров, где k — положительное целое число.

Постройте эффективный алгоритм для расчёта максимальной общей прибыли при заданных условиях.

6.4. У вас есть строка $s[1\ldots n]$ длины n, которая предположительно составлена из русских слов, но без пробелов и пунктуации (например, «папарамама»). Вы хотите проверить это, используя словарь, доступный как функция $\texttt{dict}(w) =$ «w — существующее слово» со значениями 0 и 1.

(a) Постройте алгоритм, основанный на методе динамического программирования, который определяет, можно ли строку $s[1\ldots n]$ разбить на слова из словаря. Время работы $O(n^2)$ (считаем, что обращение к процедуре `dict` требует времени $O(1)$).

(b) Модифицируйте алгоритм так, чтобы он возвращал одно из возможных разбиений на слова, если строка оказалась корректной.

6.5. *Расстановка фишек.* Рассмотрим доску (таблицу), у которой 4 строки и n столбцов и в каждой клетке которой написано число. У нас есть большой запас фишек, и мы хотим поместить в некоторые клетки по фишке. Сумма чисел в клетках с фишками должна быть максимальной. При этом нельзя ставить фишки в клетки, граничащие по горизонтали или вертикали (в соседних по диагонали клетках фишки стоять могут).

(a) Перечислите все допустимые *конфигурации* фишек в одном столбце.

Правила игры определяют, какие конфигурации могут стоять в соседних столбцах.

(b) Постройте линейный по времени алгоритм нахождения наилучшей расстановки, используя динамическое программирование. (Подсказка: подзадачи соответствуют доскам меньшей ширины с заданной конфигурацией в последнем столбце.)

6.6. Определим операцию «умножения» над символами a, b, c следующим образом:

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Например, $ab = b$, $ba = c$. Заметим, что определённая нами операция умножения не является ни ассоциативной, ни коммутативной.

Постройте эффективный алгоритм, который получает строку, составленную из этих трёх символов (например, $bbbbac$) и определяет, можно ли в ней расставить скобки так, чтобы после всех умножений получилось a. Например, для входа $bbbbac$ алгоритм должен дать положительный ответ, так как $((b(bb))(ba))c = a$.

6.7. *Палиндром* читается одинаково слева направо и справа налево. Например, у последовательности ACGTGTCAAAATCG есть подпоследовательности-палиндромы ACGCA и AAAA (а подпоследовательность ACT палиндромом *не* является). Постройте алгоритм, получающий последовательность $x[1\ldots n]$ и возвращающий длину самой длинной подпоследовательности-палиндрома (а также саму подпоследовательность). Время работы алгоритма должно быть $O(n^2)$.

6.8. Пусть заданы две строки $x = x_1x_2\ldots x_n$ и $y = y_1y_2\ldots y_m$. Необходимо найти длину *наибольшей общей подстроки* (longest common substring), то есть наибольшее k, для которого $x_ix_{i+1}\ldots x_{i+k-1} = y_jy_{j+1}\ldots y_{j+k-1}$ при некоторых i и j. Покажите, как сделать это за время $O(mn)$.

6.9. Представим себе, что в некоторой системе программирования разрезание строки на части реализовано с помощью копирования. Оно занимает n шагов для строк длины n (независимо от позиции разреза). Мы хотим разрезать заданную строку в заданных местах (но порядок разрезов не фиксирован). Стоимость зависит от порядка: если строку длины 20 разрезать сначала в позиции 3, а потом в позиции 10, то суммарная стоимость будет равна $20 + 17 = 37$. Если же разрезать в тех же позициях, но в другом порядке, то стоимость будет меньше: $20 + 10 = 30$.

Постройте алгоритм, использующий динамическое программирование, который по длине строки и местам разрезов находит оптимальный порядок разрезания.

6.10. *Подсчёт орлов.* Пусть есть n монет и пусть $p_i \in [0, 1]$ — вероятность выпадения i-й монеты орлом. Постройте алгоритм, который по заданным $0 \leqslant k \leqslant n$ и $p_1, \ldots, p_n \in [0, 1]$ вычисляет вероятность того, что ровно k из этих монет выпадут орлом (монеты подбрасываются независимо). Время работы[1] должно быть $O(n^2)$. Считаем, что умножение и сложение чисел на отрезке $[0, 1]$ осуществляется за время $O(1)$.

6.11. Даны строки $x = x_1x_2\ldots x_n$ и $y = y_1y_2\ldots y_m$. Надо найти длину *наибольшей общей подпоследовательности* (longest common subsequence), то есть наибольшее k, для которого $x_{i_1}x_{i_2}\ldots x_{i_k} = y_{j_1}y_{j_2}\ldots y_{j_k}$ при некоторых $i_1 < i_2 < \ldots < i_k$ и $j_1 < j_2 < \ldots < j_k$. Время работы алгоритма должно быть $O(mn)$.

6.12. Выпуклый n-угольник P на плоскости задан координатами своих вершин. *Триангуляцией* (triangulation) P называется набор из $n-3$ непересекающихся диагоналей P. Триангуляция делит P на $n-2$ непересекающихся треугольника. Стоимость триангуляции — это сумма длин входящих в неё

[1] Эту задачу можно решить даже за время $O(n \log^2 n)$.

диагоналей. Постройте эффективный алгоритм для нахождения триангуляции P минимальной стоимости. (Подсказка: пронумеруйте вершины P числами от 1 до n, начав с произвольной вершины и двигаясь против часовой стрелки. Для $1 \leqslant i < j \leqslant n$ обозначьте через $A[i, j]$ минимальную стоимость триангуляции многоугольника с вершинами $i, i+1, \ldots, j$.)

6.13. На столе лежат в ряд n карт стоимостью $v_1, v_2, \ldots, v_n$ (слева направо). Два игрока по очереди берут карты, причём можно брать только крайнюю карту (с любой стороны), до тех пор, пока не кончатся все карты. Каждый из игроков заинтересован в максимальной суммарной стоимости взятых им карт (представьте себе, что это купюры). Считайте, что n чётно.

(a) Покажите, что жадная стратегия (брать ту из крайних карт, которая дороже) не оптимальна: уже первый ход по этому правилу может помешать достичь оптимума.

(b) Постройте алгоритм отыскания оптимальной стратегии для первого игрока за время $O(n^2)$. Получив на вход $v_1, \ldots, v_n$, алгоритм должен произвести некоторую «предобработку» за время $O(n^2)$, а вычисление каждого следующего хода должно выполняться за $O(1)$ шагов (с использованием сохранённой информации).

6.14. *Разрезание ткани.* Есть прямоугольный кусок ткани $X \times Y$, где X и Y — положительные целые числа. Из этой ткани можно делать n видов изделий; каждое изделие вида i использует прямоугольник $a_i \times b_i$ и приносит доход c_i (все a_i, b_i, c_i — тоже положительные целые числа). Станок для резки ткани умеет резать прямоугольные куски только вдоль их стороны (любой). Какой максимальный доход можно извлечь из куска $X \times Y$? Ваш алгоритм должен построить оптимальную последовательность разрезов. (Количество изделий не ограничивается — в частности, можно вовсе не делать изделий некоторых видов.)

6.15. Пусть две команды A и B играют в серии игр (без ничьих), пока одна из команд не выиграет n раз. Будем считать, что вероятность выигрыша 50% и игры независимы. Пусть на данный момент команда A выиграла i игр, а команда B выиграла j игр. Найдите вероятность того, что A выиграет серию. (Например, при $i = n - 1$ и $j = n - 3$ эта вероятность равна $7/8$, так как единственный проигрышный вариант — это три выигрыша B подряд.)

6.16. *Задача о распродажах* (*сообщил Лотфи Заде*). Воскресным утром в городе проходят n распродаж. Мы знаем v_i — выгоду от распродажи i, а также d_{ij} — стоимость поездки от i к j. Известны также стоимости d_{0j} и d_{j0} поездки от дома до распродажи j и обратно. Нужно выбрать наиболее выгодный маршрут, начинающийся и заканчивающийся у дома и позволяющий посетить некоторое число распродаж. Выгода определяется как разность между общей выгодой от посещённых распродаж и стоимостью всех поездок.

Постройте алгоритм, который решает поставленную задачу за время $O(n^2 2^n)$. (Подсказка: вспомните задачу коммивояжёра.)

6.17. Надо заплатить v рублей, имея купюры в $x_1, x_2, \ldots, x_n$ рублей (купюр каждого типа сколько угодно). Это не всегда возможно (скажем, купюрами в 10 и 25 рублей можно заплатить 45 рублей, но не 47 и не 15).

Постройте основанный на методе динамического программирования алгоритм со временем работы $O(nv)$, выясняющий, возможна ли уплата.

6.18. Как решить предыдущую задачу (упражнение 6.17), если каждая купюра имеется в единственном экземпляре? (Например, купюрами в 1, 5, 10, 20 рублей можно заплатить $16 = 1 + 5 + 10$ и $31 = 1 + 10 + 20$ рублей, но не 40 (не хватит денег). Время работы алгоритма по-прежнему $O(nv)$.

6.19. Ещё один вариант той же задачи (упражнение 6.17): купюр каждого типа сколько угодно, но всего разрешается использовать не более k купюр. (Например, если есть купюры в 5 и 10 рублей, а $k = 6$, то 55 разменять можно, а 65 — нет).

6.20. *Оптимальное двоичное дерево поиска.* Нам известны частоты появления ключевых слов в некоторой программе, то есть дана таблица вроде такой:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

Мы хотим построить двоичное дерево поиска: в вершинах стоят ключевые слова, ключевое слово в каждой вершине лексикографически больше всех слов в левом поддереве и меньше всех слов в правом поддереве.

Слева на рис. 6.12 приведён пример сбалансированного дерева, в котором любое слово можно найти, посетив не более трёх вершин (для «while» это вершины «end», «then» и «while»). Но если слова встречаются с разной частотой, разумно минимизировать *среднее количество сравнений* для поиска слова. Для дерева слева оно равно

$$\mathtt{cost} = 1 \cdot (0{,}04) + 2 \cdot (0{,}40 + 0{,}10) + 3 \cdot (0{,}05 + 0{,}08 + 0{,}10 + 0{,}23) = 2{,}42.$$

Оптимальное с этой точки зрения дерево показано справа ($\mathtt{cost} = 2{,}18$).

Постройте эффективный алгоритм для следующей задачи.

Вход: список из n слов в алфавитном порядке; частоты их появления в тексте $(p_1, p_2, \ldots, p_n)$.
Выход: двоичное дерево поиска с наименьшим средним количеством сравнений.

6.21. *Вершинным покрытием* (vertex cover) графа $G = (V, E)$ называется множество его вершин $S \subseteq V$, которое содержит хотя бы один конец каждого ребра графа. Например, на рисунке множества $\{A, B, C, D, E, F, G\}$ и $\{A, C, D, F\}$ являются вершинными покрытиями, а $\{C, E, F\}$ — нет. Размер наименьшего

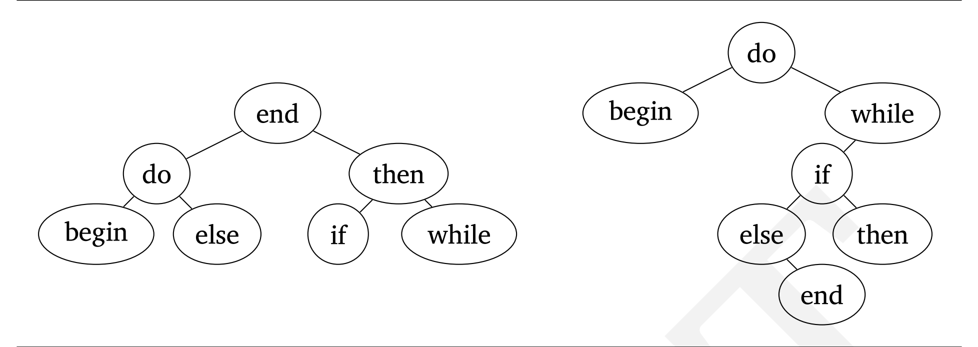
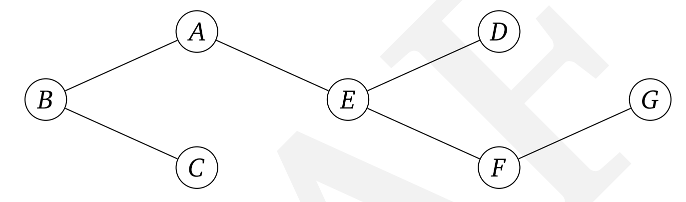


Рис. 6.12. Двоичные деревья поиска ключевых слов

вершинного покрытия равен 3 (один из вариантов: $\{B, E, G\}$).



Постройте линейный алгоритм для следующей задачи:

Вход: неориентированное дерево $T = (V, E)$.
Выход: размер наименьшего вершинного покрытия дерева T.

6.22. Постройте алгоритм со временем работы $O(nt)$ для следующей задачи.

Вход: список положительных целых чисел $a_1, a_2, \ldots, a_n$; положительное целое t.
Вопрос: можно ли представить t как сумму некоторых из $a_1, a_2, \ldots, a_n$? Каждое a_i разрешается использовать не более одного раза (можно не использовать вообще).

(Подсказка: рассмотрите подзадачи «можно ли представить s как сумму некоторых из $a_1, a_2, \ldots, a_i$».)

6.23. Обработка изделий выполняется в n шагов; шаг i выполняется машиной M_i. Машина M_i исправна с вероятностью r_i (вероятность сбоя равна $1 - r_i$). Сбои машин являются независимыми событиями, и вероятность корректной отработки всей системы равна произведению $r_1 r_2 \ldots r_n$. Для повышения надёжности можно купить m_i штук машины M_i; нам достаточно, чтобы хотя бы одна из них сработала. Вероятность того, что все m_i дадут сбой одновременно, теперь равна $(1 - r_i)^{m_i}$, то есть вероятность корректного завершения шага составляет $1 - (1 - r_i)^{m_i}$, а вероятность правильной работы всей системы — $\prod_{i=1}^{n}(1 - (1 - r_i)^{m_i})$. Стоимость одной машины M_i равна c_i, а общий бюджет на покупку всех машин равен B (c_i и B являются положительными

целыми числами). При заданных параметрах B, $r_1, \ldots, r_n$ и $c_1, \ldots, c_n$ найдите вариант с максимальной вероятностью корректной работы.

6.24. *Время и память в динамическом программировании.* Рассмотренный нами алгоритм вычисления расстояния редактирования строк длины m и n заполняет таблицу размера $O(mn)$. При больших m и n такому алгоритму просто не хватит памяти. Как можно обойтись меньшим объёмом памяти?

(a) Допустим, что нас интересует только расстояние редактирования, но не соответствующее оптимальное выравнивание. Покажите, что тогда в каждый момент не нужно хранить всю таблицу и можно обойтись объёмом памяти $O(n)$.

(b) Теперь допустим, что мы хотим найти и оптимальное выравнивание. Как мы видели, это эквивалентно поиску кратчайшего пути из вершины $(0, 0)$ в вершину (n, m) в соответствующем графе. Любой такой путь должен проходить через вершину $(k, m/2)$ для некоторого k. Модифицируйте алгоритм поиска расстояния редактирования, чтобы он заодно выдавал и k. (Считайте, что m есть степень двойки.)

(c) Рассмотрим теперь следующий рекурсивный алгоритм.

функция FindPath($(0, 0) \to (n, m)$)

вычислить k (см. выше)
FindPath($(0, 0) \to (k, m/2)$)
FindPath($(k, m/2) \to (n, m)$)
вернуть объединение найденных двух путей

Покажите, что по данной схеме алгоритм можно реализовать так, чтобы время его работы было $O(nm)$, а память — $O(n)$.

6.25. Рассмотрим следующую задачу о разбиении на три части (3-partition). Необходимо разбить набор целых чисел $a_1, \ldots, a_n$ на три части так, чтобы суммы чисел в каждой из частей были одинаковы. Другими словами, мы хотим разбить множество $\{1, \ldots, n\}$ на три части I, J, K так, что

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i.$$

Например, набор $(1, 2, 3, 4, 4, 5, 8)$ разбить можно: $(1, 8)$, $(4, 5)$, $(2, 3, 4)$, а набор $(2, 2, 3, 5)$ нельзя.

Постройте алгоритм, основанный на методе динамического программирования, со временем работы, зависящим полиномиально от n и $\sum_i a_i$.

6.26. *Выравнивание последовательностей.* Чтобы понять, за что отвечает новый найденный ген, в базе данных ищут похожие гены. Критерием похожести является то, насколько хорошо они могут быть *выровнены*. Будем считать гены словами в алфавите $\Sigma = \{A, C, G, T\}$. Выравниванием генов $x = \text{ATGCC}$ и $y = \text{TACGCA}$ будет их расположение друг над другом:

–	A	T	–	G	C	C
T	A	–	C	G	C	A

Символ «—» обозначает пропуск (gap). Порядок сохраняется, и каждый столбец должен содержать либо два символа, либо символ и пропуск. Качество выравнивания определяется таблицей стоимостей замен размера $(|\Sigma|+1)\times(|\Sigma|+1)$ (дополнительные строка и столбец соответствуют пропуску). Например, качество рассмотренного выше выравнивания равно

$$\sigma(-,\mathrm{T})+\sigma(\mathrm{A},\mathrm{A})+\sigma(\mathrm{T},-)+\sigma(-,\mathrm{C})+\sigma(\mathrm{G},\mathrm{G})+\sigma(\mathrm{C},\mathrm{C})+\sigma(\mathrm{C},\mathrm{A}).$$

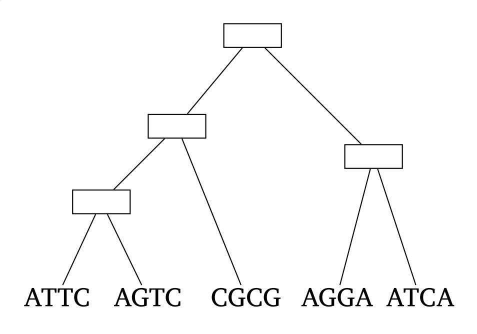
Постройте алгоритм, использующий динамическое программирование, который по входным словам $x[1\ldots n]$ и $y[1\ldots m]$ и таблице стоимостей σ находит за время $O(mn)$ наилучшее выравнивание.

6.27. *Выравнивания со штрафом за пропуски.* Биологический смысл пропусков при поиске оптимального выравнивания (упражнение 6.26) — ошибки, возникающие при копировании ДНК. При таком копировании могут пропускаться скорее целые куски, чем отдельные буквы, так что штраф за пропуск длины 10 должен быть значительно меньше, чем десятикратный штраф за один пропуск. Положим штраф за пропуск длины k равным c_0+c_1k, где c_0 и c_1 — константы (и $c_0>c_1$). Как тогда искать наилучшее выравнивание?

6.28. *Локальное выравнивание последовательностей.* Часто две последовательности ДНК сильно различаются, но при этом содержат куски, которые очень похожи. Постройте алгоритм, который получает на вход две строки $x[1\ldots n]$ и $y[1\ldots m]$ и матрицу стоимостей замен σ (см. упражнение 6.26) и находит подстроки x' и y' строк x и y, максимально похожие друг на друга в описанном смысле. Время работы алгоритма должно быть $O(mn)$.

6.29. *Куски генома.* При поиске участков генома, похожих на данный ген, может быть полезна следующая задача. Пусть дан набор «частичных совпадений» в геноме длины n, то есть троек (l_i, r_i, w_i), где $1\leqslant l_i\leqslant r_i\leqslant n$ задают отрезок совпадения $[l_i, r_i]$, а w_i — вес совпадения. Необходимо выбрать из него семейство частичных совпадений с непересекающимися отрезками и максимальным суммарным весом.

6.30. *Восстановление деревьев эволюции.* Если в геноме для нескольких видов встречаются разные варианты одного и того же гена, можно пытаться реконструировать дерево эволюции, исходя из того, что гены менялись постепенно вдоль этого дерева. Будем считать, что для каждого вида имеется вариант гена, представляющий собой строку длины k в алфавите $\Sigma=\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$.



Историю эволюции будем представлять деревом, в листьях которого стоят виды, а внутренние вершины соответствуют их общим эволюционным предкам. Например, на рисунке представлен возможный ход эволюции, когда общий вид-предок разделился на два, каждый из них тоже разделился на два, а из полученных четырёх видов один (самый левый на рисунке) также разделился.

Принцип максимальной простоты предлагает искать дерево и строки в его вершинах, при которых общее количество изменений вдоль всех рёбер минимально. Такое дерево искать довольно сложно, и мы рассмотрим более простой вариант задачи: пусть структура дерева уже известна, а надо найти строки $s(u)$ для всех внутренних вершин u (варианты гена для эволюционных предков). Это тоже строки длины k. При этом надо минимизировать величину

$$\texttt{score}(T) = \sum_{(u,v)\in E(T)} (\text{число позиций, в которых } s(u) \text{ и } s(v) \text{ различаются}).$$

(a) В приведённом выше примере есть несколько оптимальных вариантов; укажите один из них.

(b) Постройте эффективный (в смысле оценки времени как функции n и k) алгоритм для данной задачи. (Подсказка: буквы в разных позициях можно обрабатывать независимо.)

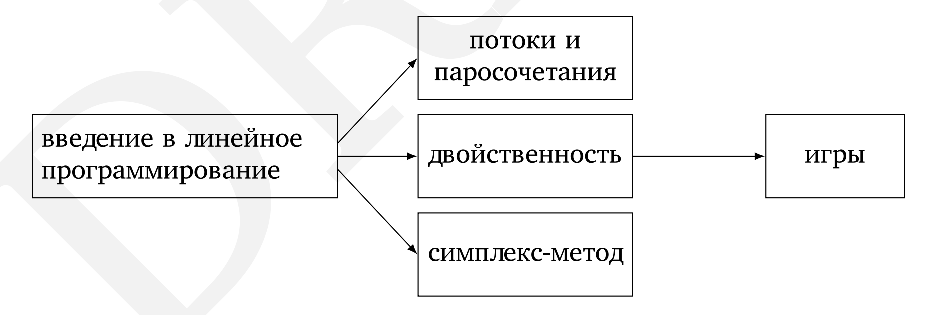
Глава 7
Линейное программирование и сводящиеся к нему задачи

Мы уже встречались с разными задачами *оптимизации*, когда искали *кратчайший* путь, *минимальное* покрывающее дерево, *наибольшую* возрастающую подпоследовательность и так далее. В таких задачах мы ищем решение, которое

- удовлетворяет определённым ограничениям (путь должен проходить по рёбрам графа и вести из s в t, дерево должно содержать все вершины графа, подпоследовательность должна быть возрастающей);
- является наилучшим среди всех таких решений (согласно выбранному критерию).

Задача линейного программирования возникает, если ограничения задаются *линейными неравенствами*, а целевая функция (которую надо сделать максимальной или минимальной) *линейна*. К такому виду можно привести много разных задач.

Глава разделена на несколько частей, которые можно читать по отдельности, учитывая следующие зависимости:



7.1. Введение в линейное программирование

В задаче линейного программирования (ЛП) есть набор переменных, и нужно присвоить им вещественные значения,

- соблюдая ограничения, представляющие собой набор линейных уравнений или неравенств с этими переменными, и
- максимизируя или минимизируя (при этих ограничениях) данную линейную целевую функцию.

7.1.1. Пример: максимизация прибыли

Кондитерская производит два сорта шоколада — А и Б. Плитка шоколада А стоит 1 тугрик, а шоколада Б — 6 тугриков. Сколько плиток сортов А и Б (обозначим эти числа x_1 и x_2) нужно ежедневно производить, чтобы максимизировать прибыть?

Ограничения (помимо очевидных $x_1, x_2 \geqslant 0$):

(1) ежедневный спрос ограничен 200 плитками А и 300 плитками Б;

(2) ежедневное производство не больше 400 плиток (всего) в день.

Каковы оптимальные объёмы производства?

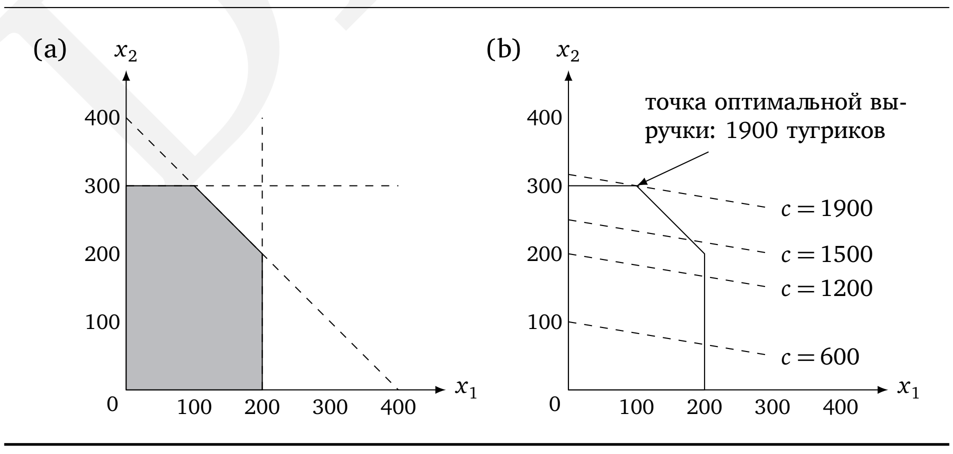
Эту задачу можно записать так:

$$\begin{array}{ll} \text{целевая функция} & x_1 + 6x_2 \to \max \\ & x_1 \leqslant 200 \\ \text{ограничения} & x_2 \leqslant 300 \\ & x_1 + x_2 \leqslant 400 \\ & x_1, x_2 \geqslant 0 \end{array}$$

Такая запись называется «задачей линейного программирования» или «линейной программой» (буквальный перевод английского термина linear program); в последнее время второй вариант используется довольно часто, несмотря на то что по-русски это звучит странно.

Значения (x_1, x_2) определяют точку на координатной плоскости; линейное уравнение задаёт прямую, а ограничение в виде линейного неравенства — полуплоскость. Чтобы учесть все ограничения, надо найти пересечение этих полуплоскостей, и получится множество всех *допустимых решений* (feasible solutions) этой задачи, в данном случае выпуклый многоугольник (рис. 7.1).

Рис. 7.1. (a) Множество допустимых решений. (b) Показаны прямые $x_1 + 6x_2 = c$, задающие точки с выручкой c, для нескольких значений c.



В этом многоугольнике мы ищем точку, в которой целевая функция (выручка) максимальна. Точки с выручкой в c рублей лежат на прямой $x_1 + 6x_2 = c$, которая имеет угловой коэффициент $-1/6$ (см. рис. 7.1). При увеличении c эта «прямая выручки» сдвигается вверх параллельно самой себе. Мы хотим максимизировать c, поэтому должны сдвинуть прямую (сохраняя пересечение с многоугольником) как можно дальше. На рисунке видно искомое положение прямой: она пересекает многоугольник в вершине (и если ещё подвинуть, пересечение пропадёт). Если бы коэффициенты в целевой функции были бы другими, вместо оптимальной точки могло бы получиться целое оптимальное ребро: это означало бы, что есть целый отрезок вариантов, где достигается оптимум (и крайними из них были бы вершины).

Если максимум достигается, то (как мы увидим), он достигается в некоторой вершине. Но максимума может не быть, и даже по двум причинам:

1) ограничения настолько сильные, что все их выполнить невозможно, например,

$$x \leqslant 1, \quad x \geqslant 2;$$

в этом случае говорят, что линейная программа *несовместна* или *невыполнима* (infeasible);

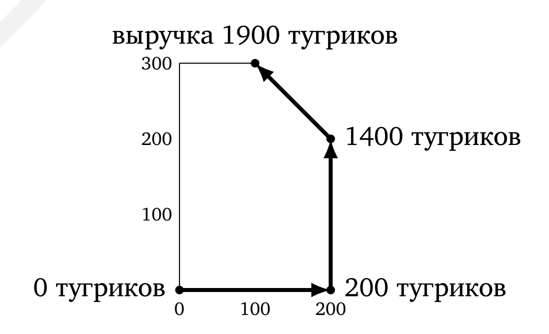
2) ограничения настолько слабые, что возможны произвольно большие значения целевой функции; например, так будет для программы

$$x_1 + x_2 \to \max$$
$$x_1, x_2 \geqslant 0$$

В этом случае говорят о *неограниченной* (unbounded) линейной программе.

Решение задач линейного программирования

Говоря о решении задач линейного программирования, мы имеем в виду поиск точки оптимума. Это можно делать с помощью *симплекс-метода* (simplex method), изобретённого Джорджем Данцигом в 1947 году. Мы изложим его в разделе 7.6, в общих чертах его можно описать так: алгоритм начинает работу в вершине — в нашем случае это может быть (0, 0) — и на каждом шаге пытается перейти по ребру в соседнюю вершину с лучшим целевым значением. Вот возможная траектория в нашем примере:



Симплекс-метод завершает работу и объявляет текущую вершину оптимальной, если лучшего соседа нет. Почему эта *локальная* проверка гарантирует *глобальную* (не только среди соседей) оптимальность? Представьте себе прямую выручки, проходящую через данную вершину. Если все соседи вершины лежат ниже этой прямой, то и остальная часть (выпуклого) многоугольника допустимых решений тоже будет ниже.

Дополнительные товары

Не останавливаясь на достигнутом, кондитерская начинает производство третьего сорта шоколада (В) «премиум-класса», по 13 тугриков за плитку. Тогда ежедневный выпуск описывается тройкой чисел x_1, x_2, x_3 (для трёх сортов). Прежние ограничения на x_1 и x_2 сохраняются, но в ограничение на общий выпуск теперь включается и x_3: сумма всех трёх переменных не может превышать 400. Кроме того, оказывается, что упаковка для В втрое более трудоёмкая, чем для Б, и это налагает ещё одно ограничение $x_2 + 3x_3 \leqslant 600$. Каковы оптимальные объёмы производства теперь?

Возникает задача:

$$\begin{aligned} x_1 + 6x_2 + 13x_3 &\to \max \\ x_1 &\leqslant 200 \\ x_2 &\leqslant 300 \\ x_1 + x_2 + x_3 &\leqslant 400 \\ x_2 + 3x_3 &\leqslant 600 \\ x_1, x_2, x_3 &\geqslant 0 \end{aligned}$$

В ней пространство решений трёхмерное. Линейное уравнение задаёт плоскость в трёхмерном пространстве, а неравенство — полупространство по одну сторону от плоскости. Множество допустимых решений — пересечение семи полупространств — является многогранником (рис. 7.2; для наглядности масштаб по оси x_3 на рисунке увеличен). Можете ли вы определить по рисунку, какое неравенство к какой грани относится?

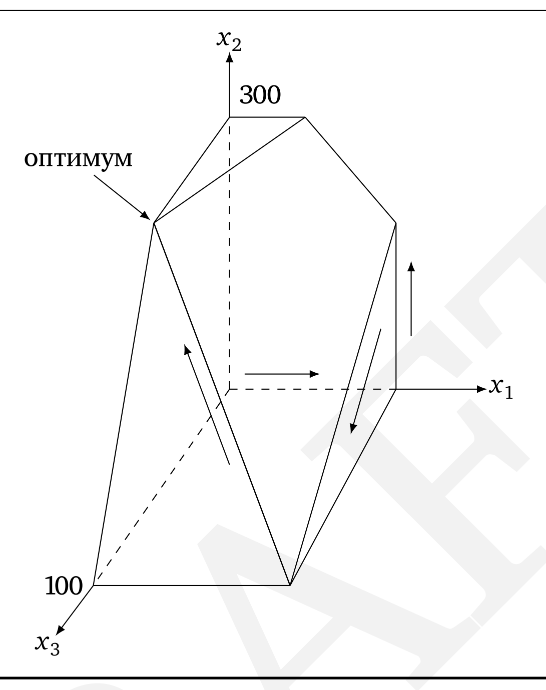
Выручка c соответствует плоскости $x_1 + 6x_2 + 13x_3 = c$. При увеличении c плоскость выручки сдвигается параллельно самой себе (в сторону увеличения переменных), пока не перестанет задевать множество допустимых решений. Последней точкой пересечения будет вершина (0, 300, 100) с оптимальной выручкой 3100 тугриков.

Как тут работает симплекс-метод? Как и раньше, он переходит от вершины к вершине по рёбрам многогранника, каждый раз увеличивая выручку. Возможная траектория показана стрелками на рисунке 7.2 и соответствует следующей последовательности вершин (в квадратных скобках указан доход в этой вершине):

$$\underset{[0]}{(0,0,0)} \to \underset{[200]}{(200,0,0)} \to \underset{[1400]}{(200,200,0)} \to \underset{[2800]}{(200,0,200)} \to \underset{[3100]}{(0,300,100)}$$

Дойдя до вершины, где нет лучшего соседа, алгоритм останавливается и сообщает, что достиг максимума. Из геометрических соображений ясно, что

Рис. 7.2. Многогранник допустимых решений для задачи линейного программирования с тремя переменными.



если все соседи вершины лежат по одну сторону от плоскости выручки, там же должен лежать и весь многогранник.

Немного магии

Убедиться, что точка $(0, 300, 100)$ с общей выручкой 3100 оптимальна, можно так: сложите второе неравенство с третьим и прибавьте к ним четвёртое, умноженное на 4. Получится неравенство $x_1 + 6x_2 + 13x_3 \leqslant 3100$, показывающее, что выручка не может превысить 3100, так что наше решение оптимально.

Но откуда же мы взяли загадочные коэффициенты $(0, 1, 1, 4)$, с которыми надо складывать четыре неравенства?

В разделе 7.4 мы увидим, что их можно получить, решив другую задачу линейного программирования, называемую *двойственной* к исходной.

Что будет, если добавить четвёртый сорт шоколада (или ещё сотни сортов)? Картинку тут уже не нарисуешь, но симплекс-метод по-прежнему работает (см. раздел 7.6). Есть программные продукты, реализующие симплекс-метод (и заботящиеся о всяких подробностях, типа ошибок округления), так что на практике бывает достаточно представить задачу в виде задачи линейного программирования.

7.1.2. Пример: планирование производства

Рассмотрим более запутанный пример и убедимся, что и здесь задача сводится к линейному программированию. Пусть наша компания производит ковры ручной работы. Спрос на них в разные месяцы разный. Мы планируем работу на следующий год и исходим из оценки спроса для всех месяцев: это двенадцать чисел $d_1, d_2, \ldots, d_{12}$ в диапазоне от 440 до 920.

В начале года в компании 30 работников, каждый из которых делает по 20 ковров в месяц и получает месячную зарплату 2000 рублей; начальных запасов ковров у нас нет.

Как мы можем реагировать на колебания спроса? Имеются три способа:

1. *Сверхурочная работа* требует оплаты на 80% больше обычной. Кроме того, сверхурочная работа должна составлять не более 30% основной.

2. *Набор и увольнение работников* обходятся в 320 рублей и 400 рублей соответственно на каждого работника.

3. *Хранение продукции* требует 8 рублей на каждый ковёр ежемесячно. В начале и конце года склад должен быть пуст.

Для простоты мы считаем, что работники принимаются и увольняются в начале месяца, что покупатели приходят за коврами к концу месяца и что хранение оплачивается по складским остаткам на конец каждого месяца.

Эта достаточно сложная задача может быть сформулирована в виде линейной программы, надо только правильно выбирать переменные. Положим:

$$\begin{aligned}
w_i &= \text{количество работников в } i\text{-й месяц } (i=1,\ldots,12);\ w_0=30;\\
x_i &= \text{количество ковров, сделанных за } i\text{-й месяц;}\\
o_i &= \text{количество ковров, сделанных сверхурочно за } i\text{-й месяц;}\\
h_i, f_i &= \text{количество рабочих, нанятых и уволенных в начале } i\text{-го месяца;}\\
s_i &= \text{количество ковров на складе в конце } i\text{-го месяца; } s_0=0.
\end{aligned}$$

Всего получается 74 переменных (считая w_0 и s_0). Теперь запишем ограничения. Все переменные должны быть неотрицательными:

$$w_i, x_i, o_i, h_i, f_i, s_i \geqslant 0, \quad i=1,\ldots,12.$$

Общее число ковров, сделанных за месяц (в основное и сверхурочное время):

$$x_i = 20w_i + o_i.$$

Сверхурочное производство ограничено:

$$o_i \leqslant 6w_i.$$

Число рабочих может меняться в начале каждого месяца:

$$w_i = w_{i-1} + h_i - f_i.$$

Количество ковров на складе в конце месяца — это их количество в начале, плюс произведённое, минус спрос за месяц:

$$s_i = s_{i-1} + x_i - d_i.$$

Все эти ограничения должны выполняться при $i = 1, \ldots, 12$. В конце года склад должен быть пуст:

$$s_{12} = 0.$$

Наконец, какова целевая функция? Нам надо минимизировать расходы[1]:

$$2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i \to \min .$$

После того как всё это записано, мы можем найти оптимальную стратегию производства за доли секунды с помощью симплекс-метода.

Тут есть, однако, важная деталь. Оптимальное решение может оказаться *дробным* («И вышло у меня в ответе: // два землекопа и две трети»). Если программа велит нанять 10,6 работников, то 10,6 придётся округлить или до 10, или до 11, и решение уже будет не совсем оптимальным (или даже нарушатся ограничения). В нашем примере большинство чисел двузначные, так что округление их не сильно меняет (а небольшие отклонения в жизни всё равно неизбежны), но так бывает не всегда.

Эта проблема (нужны целые решения, а легко найти вещественные) принципиальна: как мы увидим в главе 8, поиск *целочисленного* оптимума в задаче линейного программирования — важная, но очень сложная задача.

7.1.3. Пример: распределение пропускной способности

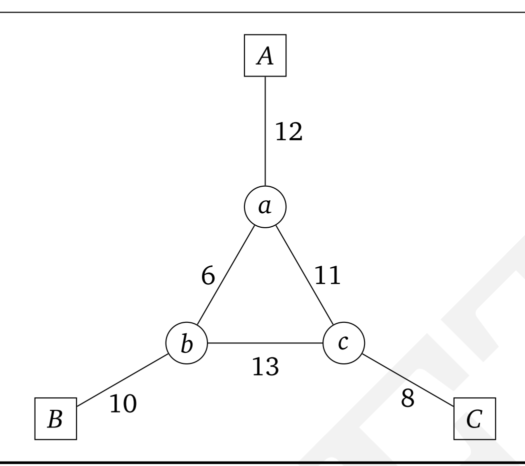
У нас есть сеть передачи данных; пропускные способности участков показаны на рисунке 7.3. Нужно установить три соединения: между A и B, между B и C, и между A и C. Каждое из них требует по крайней мере двух единиц пропускной способности, но готово взять (и оплатить) и больше (сколько дадут). Соединение A–B платит 3 рубля за единицу пропускной способности выделенного ему канала, а соединения B–C и A–C платят соответственно 2 рубля и 4 рубля.

Для каждого соединения есть два возможных пути — короткий и длинный (через третью вершину внутреннего кольца); их можно использовать одновременно в любой комбинации (скажем, пустить две единицы по короткому пути, а одну — по длинному). Как получить максимальный доход?

Составим линейную программу: для каждого заказчика (например, AB) мы обозначим через x_{AB} ширину канала по короткому пути, а через x'_{AB} — по длинному. При этом мы не должны превышать общей пропускной способности рёбер, а также должны гарантировать по две единицы для каждого соеди-

[1]В этом примере доход от продажи ковров не предусмотрен, и лишь необходимость произвести запланированное количество мешает закрыть производство и уволить всех сотрудников немедленно.

Рис. 7.3. Коммуникационная сеть между A, B и C. Числа на рёбрах — пропускные способности.



нения. Получаем такую задачу:

$$
\begin{aligned}
3x_{AB} + 3x'_{AB} + 2x_{BC} + 2x'_{BC} + 4x_{AC} + 4x'_{AC} &\to \max \\
x_{AB} + x'_{AB} + x_{BC} + x'_{BC} &\leqslant 10 && [\text{ребро } (b, B)] \\
x_{AB} + x'_{AB} + x_{AC} + x'_{AC} &\leqslant 12 && [\text{ребро } (a, A)] \\
x_{BC} + x'_{BC} + x_{AC} + x'_{AC} &\leqslant 8 && [\text{ребро } (c, C)] \\
x_{AB} + x'_{BC} + x'_{AC} &\leqslant 6 && [\text{ребро } (a, b)] \\
x'_{AB} + x_{BC} + x'_{AC} &\leqslant 13 && [\text{ребро } (b, c)] \\
x'_{AB} + x'_{BC} + x_{AC} &\leqslant 11 && [\text{ребро } (a, c)] \\
x_{AB} + x'_{AB} &\geqslant 2 \\
x_{BC} + x'_{BC} &\geqslant 2 \\
x_{AC} + x'_{AC} &\geqslant 2 \\
x_{AB}, x'_{AB}, x_{BC}, x'_{BC}, x_{AC}, x'_{AC} &\geqslant 0
\end{aligned}
$$

Даже простой пример вроде этого трудно решить вручную (попробуйте!), но симплекс-метод легко находит оптимум:

$$x_{AB} = 0,\ \ x'_{AB} = 7,\ \ x_{BC} = x'_{BC} = 1{,}5,\ \ x_{AC} = 0{,}5,\ \ x'_{AC} = 4{,}5.$$

Числа не целые, но здесь это и не нужно. Видно, что каждое ребро, за исключением a–c, используется на полную мощность.

Предостережение: мы ввели по переменной для каждого возможного пути между соединяемыми точками. С ростом сети число таких путей растёт экспоненциально, так что лучше выбирать переменные иначе (см. раздел 7.2).

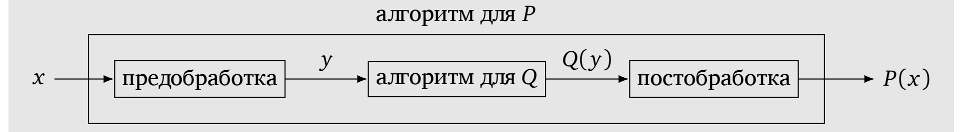
Вопрос для самоконтроля: изменится ли оптимальный вариант, если не требовать как минимум двух единиц пропускной способности для каждого соединения?

Сведе́ния

Бывает так, что одна вычислительная задача сводится к другой: любой алгоритм, решающий вторую, можно приспособить для решения первой. Например, в главе 6 мы видели, что алгоритм поиска максимального пути в ориентированном ациклическом графе, как ни странно, можно использовать для поиска максимальных возрастающих подпоследовательностей. Говорят, что задача поиска максимальной возрастающей подпоследовательности *сводится* к задаче поиска пути максимального веса в ориентированном ациклическом графе. В свою очередь, задача поиска пути максимального веса сводится к задаче поиска пути минимального веса:

```
функция LONGESTPATH(G)
изменить знак веса всех рёбер G на противоположный
вернуть SHORTESTPATH(G)
```

Общая схема: получив вход x для задачи P, мы преобразуем его в некоторый вход y для задачи Q — с тем расчётом, чтобы решение $Q(y)$ этой задачи потом можно было преобразовать в искомое решение $P(x)$ для исходной задачи P.



(Видите ли вы, каким образом сведение P = LONGEST PATH к Q = = SHORTEST PATH соответствует этой схеме?) Если процедуры пред- и постобработки работают быстро, то мы получаем быстрый алгоритм для P из *любого* быстрого алгоритма для Q!

Многие рассматриваемые нами задачи (в том числе и задача линейного программирования) важны именно потому, что к ним сводится много других задач.

7.1.4. Виды линейных программ

Линейные программы бывают разными:

1. Можно искать максимум или минимум.

2. Ограничения могут быть заданы равенствами или неравенствами.

3. Переменные могут быть неотрицательными или произвольными.

Все эти варианты сводятся друг к другу при помощи нехитрых преобразований.

1. Чтобы заменить минимум на максимум, надо изменить знак у всех коэффициентов целевой функции.

2a. Чтобы превратить ограничение-неравенство $\sum_{i=1}^{n} a_i x_i \leqslant b$ в уравнение, нужно ввести вспомогательную переменную (slack variable) $s \geqslant 0$ и написать

$$\sum_{i=1}^{n} a_i x_i + s = b,$$
$$s \geqslant 0.$$

В самом деле, исходное неравенство говорит, что переменная s, определяемая равенством, неотрицательна.

Заметим, что совсем удалить неравенства (включая ограничения $s \geqslant 0$) не удаётся (многогранник равенствами не задать).

2b. Равенство $A = B$ можно заменить на два неравенства $A \leqslant B$ и $A \geqslant B$.

3. Если мы хотим иметь дело только с неотрицательными переменными, то (неограниченную) переменную x можно всюду заменить на разность $x^+ - x^-$ двух новых неотрицательных переменных x^+, x^- (эта разность может принимать любые значения). Найдя решение новой задачи, вычисляем значение старой переменной x как $x^+ - x^-$.

Таким образом мы можем свести любую задачу линейного программирования (с максимумом или минимумом, с неравенствами и уравнениями, с неотрицательными и неограниченными переменными) к задаче в *стандартном виде*, в которой все переменные неотрицательные, ограничения являются уравнениями, а целевая функция минимизируется.

Например, нашу задачу можно переписать так:

$$\begin{array}{rcr} x_1 + 6x_2 \to \max & & -x_1 - 6x_2 \to \min \\ x_1 \leqslant 200 & & x_1 + s_1 = 200 \\ x_2 \leqslant 300 & \Rightarrow & x_2 + s_2 = 300 \\ x_1 + x_2 \leqslant 400 & & x_1 + x_2 + s_3 = 400 \\ x_1, x_2 \geqslant 0 & & x_1, x_2, s_1, s_2, s_3 \geqslant 0 \end{array}$$

Есть и другие естественные классы линейных программ, к которым можно свести любую задачу линейного программирования. Например, можно избавиться от равенств (заменив равенство $a = b$ на пару неравенств $a \leqslant b$ и $-a \leqslant -b$) и считать, что мы ищем максимум. Такую задачу удобно записать в матричном виде (см. врезку).

7.2. Потоки в сетях

7.2.1. Поставки нефти

На рисунке 7.4(a) показан ориентированный граф, представляющий сеть нефтепроводов. Цель — доставить как можно больше нефти из *истока* s в *сток* t. Для каждого нефтепровода известна его максимальная *пропускная*

Матричная запись

Линейная функция, например $x_1 + 6x_2$, может быть записана как скалярное произведение двух векторов

$$\mathbf{c} = \begin{pmatrix} 1 \\ 6 \end{pmatrix} \quad \text{и} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

обозначаемое как $\mathbf{c} \cdot \mathbf{x}$ или $\mathbf{c}^T\mathbf{x}$. Аналогично, линейные ограничения можно записать с помощью матриц:

$$\begin{aligned} x_1 \qquad &\leqslant 200 \\ x_2 &\leqslant 300 \\ x_1 + x_2 &\leqslant 400 \end{aligned} \Rightarrow \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} \leqslant \underbrace{\begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}}_{\mathbf{b}}.$$

Каждая строка матрицы $\mathbf{A}$ соответствует одному ограничению: её скалярное произведение с $\mathbf{x}$ не должно превышать значения в соответствующей позиции столбца $\mathbf{b}$. Понимая знак неравенства для векторов как сравнение по каждой координате, получаем условие $\mathbf{Ax} \leqslant \mathbf{b}$. Оно означает, что

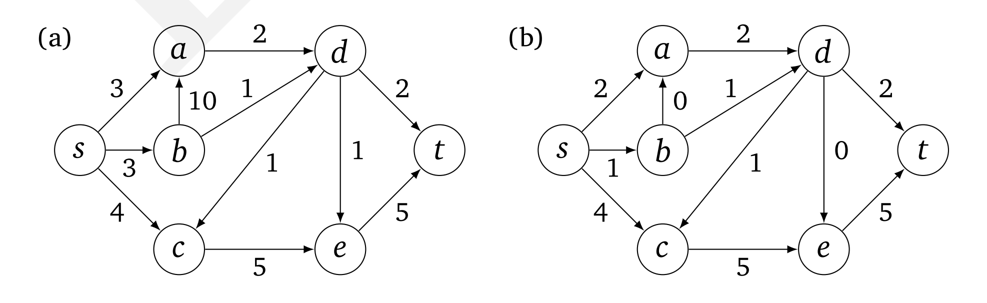
$$\mathbf{a}_i \cdot \mathbf{x} \leqslant b_i \quad \text{для всех} \quad i = 1, \ldots, m,$$

где $\mathbf{a}_1, \ldots, \mathbf{a}_m$ — строки матрицы $\mathbf{A}$. В матричных обозначениях линейная программа без ограничений-равенств и с неотрицательными переменными имеет вид

$$\begin{aligned} \mathbf{c}^T\mathbf{x} &\to \max, \\ \mathbf{Ax} &\leqslant \mathbf{b}, \\ \mathbf{x} &\geqslant 0. \end{aligned}$$

способность (максимальное количество проходящей за единицу времени нефти). На рисунке 7.4(b) показан возможный *поток* из s в t, который в сумме составляет 7 единиц. Можно ли прокачать больше?

Рис. 7.4. (a) Сеть и её пропускные способности. (b) Поток в сети.



7.2.2. Максимизация потока

Будем называть *сетью* (network) ориентированный граф $G=(V, E)$, в котором выделены *исток* (source) s и *сток* (sink) t и для каждого ребра $e \in E$ указана *пропускная способность* (capacity) $c_e > 0$.

Мы хотим перекачивать нефть из s в t, не превышая пропускной способности рёбер. Схема поставок называется *потоком* (flow): чтобы её задать, для каждого ребра e надо указать число f_e (сколько нефти по нему идёт). Ограничения:

1. Не должны быть превышены пропускные способности рёбер: $0 \leqslant f_e \leqslant c_e$ для всех $e \in E$.

2. Для любой вершины u, кроме s и t, входящий поток равен исходящему:

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}.$$

Это свойство обычно называют *сохранением потока* (flow conservation), хотя точнее было бы говорить о сохранении нефти.

Сохранение потока гарантирует, что вся выходящая из s нефть доходит до t; её количество называют *величиной* (size) потока:

$$\mathrm{size}(f) = \sum_{(s,u)\in E} f_{su} = \sum_{(u,t)\in E} f_{ut}.$$

Получаем задачу линейного программирования: найти значения $\{f_e \mid e \in E\}$, которые удовлетворяют набору линейных ограничений и максимизируют величину потока (линейную целевую функцию). *Задача о максимальном потоке сводится к задаче линейного программирования.*

Например, для сети рис. 7.4 нужно 11 переменных s_{xy}, по одной на каждое ребро $x \to y$. Ищется максимум $f_{sa}+f_{sb}+f_{sc}$ при 27 ограничениях: 11 на неотрицательность (например, $f_{sa} \geqslant 0$), 11 на пропускную способность (например, $f_{sa} \leqslant 3$) и 5 на сохранение потока (например, $f_{sc}+f_{dc}=f_{ce}$; по одному на каждую вершину графа, за исключением s и t). Остаётся запустить какую-нибудь программу, реализующую симплекс-метод: она легко установит, что поток размера 7 действительно оптимален.

7.2.3. Увеличение потока

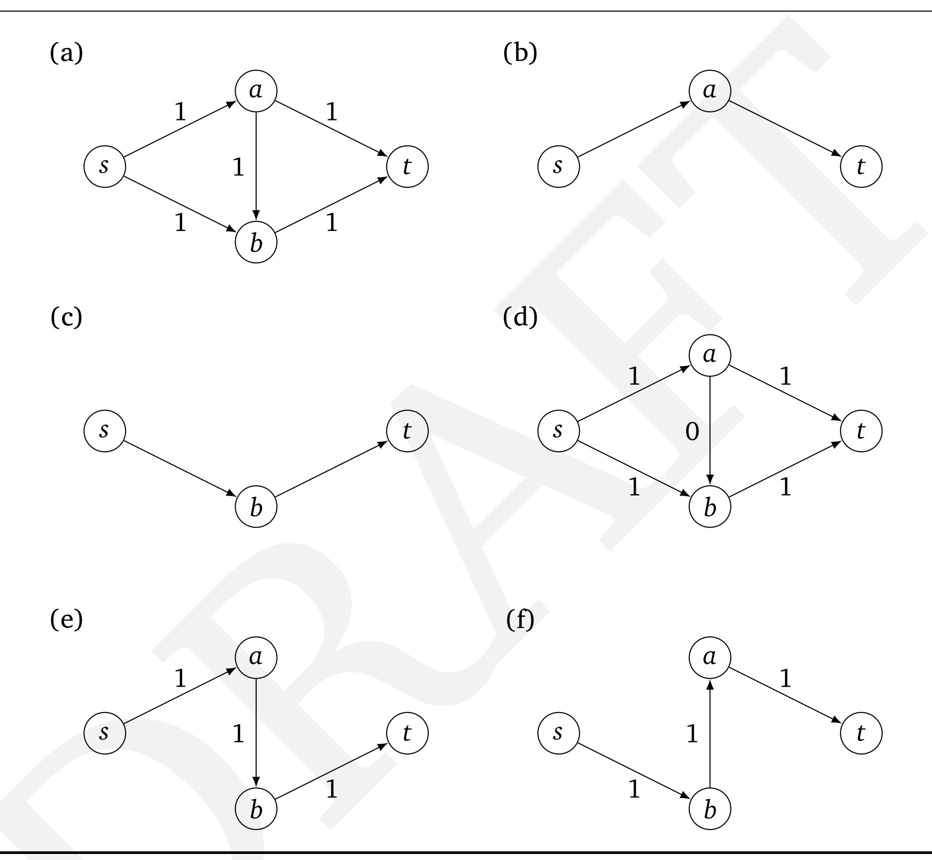
Говоря о симплекс-методе, мы упоминали, что он постепенно находит всё лучшие и лучшие решения и в конце концов приходит к оптимуму. Подробнее мы рассмотрим это в разделе 7.6, а сейчас попробуем реализовать постепенное улучшение на языке потоков по такой схеме:

```
начать с нулевого потока
повторять: выбрать подходящий путь из s в t и увеличить поток
  вдоль рёбер этого пути, насколько это возможно
```

На рисунке 7.5 в сети (a) мы сначала выбираем один путь (b), по которому можно пропустить поток размером 1, а затем замечаем, что есть ещё

один (c); в итоге получаем поток (d), полностью загружающий рёбра из истока (и, следовательно, максимальный).

Рис. 7.5. Пример работы алгоритма нахождения максимального потока. (a) Пример сети. (b) Первый выбранный путь. (c) Второй выбранный путь. (d) Итоговый поток. (e) Первым можно выбрать и такой путь. (f) Тогда на втором шаге придётся использовать такой.



Но тут есть трудность: вдруг мы на первом шаге выберем другой путь, как на рис. 7.5(e)? Он даёт только одну единицу потока и на первый взгляд блокирует все другие пути. Заметим, однако, что новые пути могут идти навстречу уже имеющемуся потоку, тем самым *сокращая* его. Например, путь на рис. 7.5(f) идёт из b в a, что реально означает прекращение потока из a в b.

В общем случае наш алгоритм ищет путь из s в t, составленный из рёбер (u, v) двух видов:

1. Ребро (u, v) содержится в исходной сети, но не вся его пропускная способность используется.

2. Обратное ребро (v, u) содержится в исходной сети, и по нему идёт ненулевой поток.

В первом случае по ребру (u, v) можно дополнительно пропустить $c_{uv} - f_{uv}$ единиц (где f — текущий поток), а во втором случае f_{vu} единиц (сокращая целиком или частично имеющийся поток по ребру (v, u)). Можно сказать, что у нас есть *остаточная сеть* (residual network) $G^f = (V, E^f)$; эта сеть имеет два типа рёбер с остаточными пропускными способностями c^f:

$$c^f_{uv} = \begin{cases} c_{uv} - f_{uv}, & \text{если } (u, v) \in E \text{ и } f_{uv} < c_{uv}, \\ f_{vu}, & \text{если } (v, u) \in E \text{ и } f_{vu} > 0. \end{cases}$$

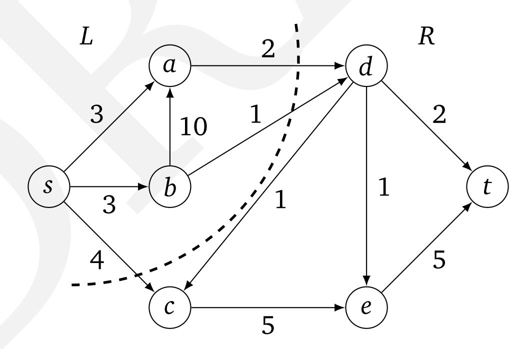
Шаг алгоритма (увеличение потока) состоит в выборе пути из истока в сток в остаточной сети. Такой путь можно найти, например, поиском в ширину, и, найдя его, мы увеличиваем поток вдоль него насколько возможно (на этом пути нас ограничивает ребро с наименьшей пропускной способностью в остаточной сети). Если в остаточной сети нет путей из s в t, алгоритм останавливается.

На рисунке 7.6 показана работа этого алгоритма (на примере задачи, с которой мы начинали); пути мы выбираем как придётся.

7.2.4. Сертификат оптимальности

Наш алгоритм, как мы вскоре увидим, не только находит оптимальный поток, но фактически предъявляет и доказательство его оптимальности. Как может выглядеть такое доказательство?

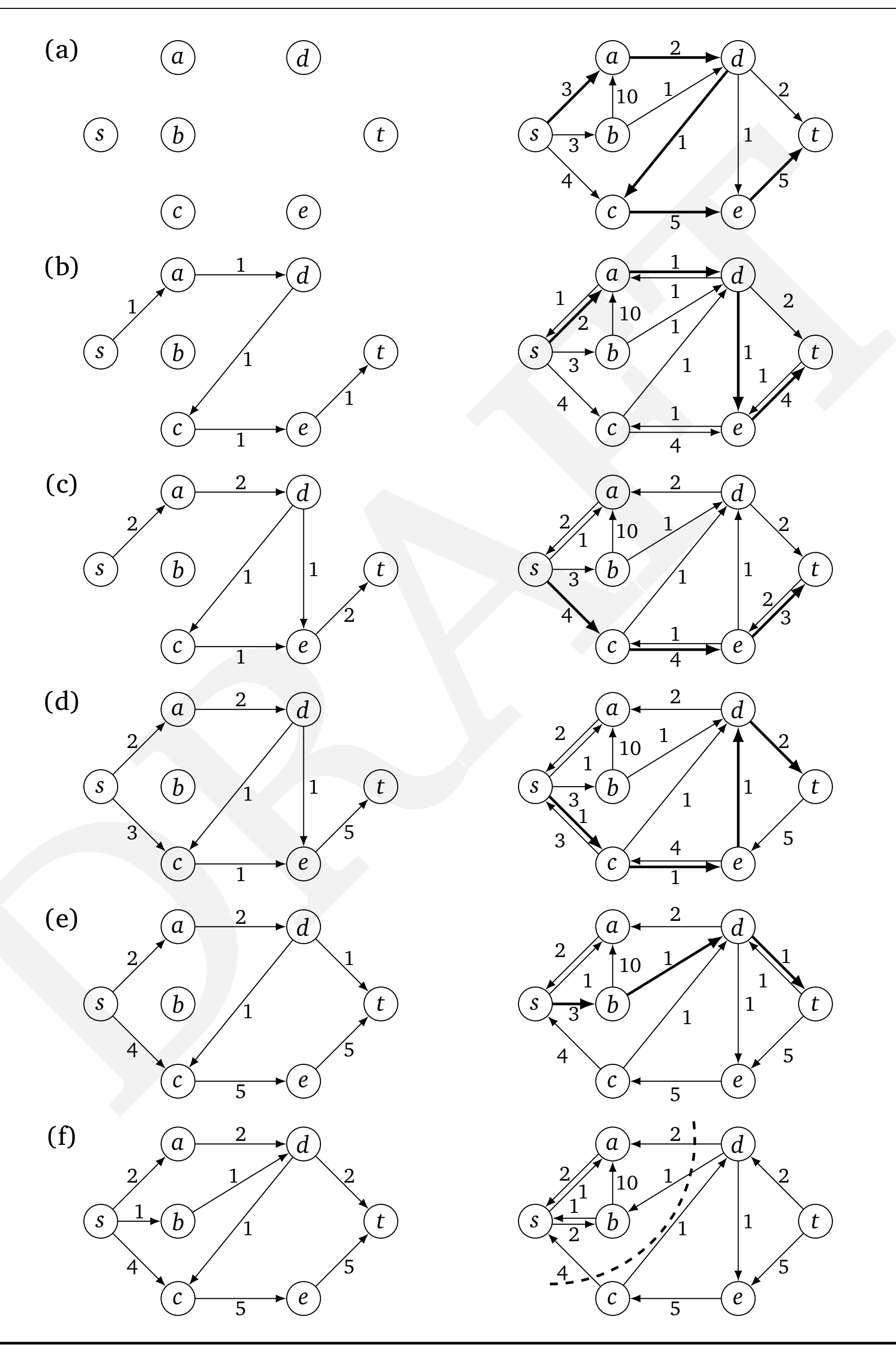
Начнём с примера. Разобьём вершины сети рис. 7.4 на две группы, как показано пунктирной линией на рисунке ($L = \{s, a, b\}$ и $R = \{c, d, e, t\}$):



Вся передаваемая нефть должна пересекать границу из L в R, и таким образом, никакой поток не может превышать суммарную пропускную способность рёбер из L в R, которая равна 7. Но это значит, что найденный нами ранее поток размера 7 оптимален! (Другой вариант разреза, который легче заметить, получается при $R = \{t\}$: даже если все входящие в вершину t рёбра загружены полностью, больше 7 не получится.)

В общем случае мы определяем (s, t)-*разрез* как разбиение всех вершин на две непересекающиеся группы L и R, содержащие соответственно s и t.

Рис. 7.6. Алгоритм поиска максимального потока, применённый к сети рис. 7.4. На каждом шаге слева показан текущий поток, а справа — остаточная сеть. Выбранный для следующего шага путь изображён жирными стрелками.



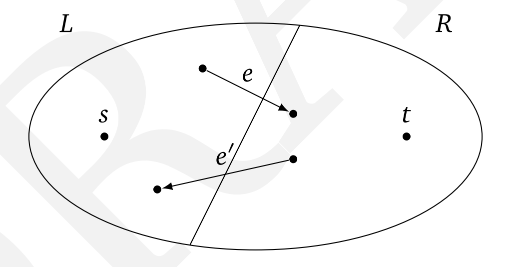
Пропускной способностью разреза называется суммарная пропускная способность рёбер из L в R; следующее очевидное наблюдение будет ключевым:

> Для любого потока f и любого (s, t)-разреза (L, R) величина потока не превышает пропускной способности разреза:
> $$\text{size}(f) \leqslant \text{capacity}(L, R).$$

Таким образом, любой разрез даёт верхнюю оценку на потоки. Некоторые разрезы имеют большие пропускные способности, и тем самым дают очень грубые оценки. Скажем, разрез $(\{s, b, c\}, \{a, d, e, t\})$ имеет пропускную способность 19. Но нам удалось предъявить разрез с пропускной способностью 7, в точности равной величине построенного потока и тем самым доказать оптимальность этого потока. Оказывается, что такой разрез существует *всегда*:

Теорема Форда—Фалкерсона о максимальном потоке и минимальном разрезе. *Размер максимального потока в сети равен пропускной способности минимального (s, t)-разреза.*

Чтобы доказать эту теорему, достаточно извлечь из нашего алгоритма искомый поток (тут и извлекать не надо) и разрез. Как это сделать? Мы знаем, что к моменту остановки вершина t больше не достижима из s в остаточной сети G^f. Обозначим через L множество вершин, которые *достижимы* из s в остаточной сети G^f, а в качестве R возьмём остальные вершины: $R = V - L$.



Мы утверждаем, что
$$\text{size}(f) = \text{capacity}(L, R).$$

В самом деле, раз в остаточной сети нет рёбер из L в R, то наш поток полностью насыщает имеющиеся в исходной сети рёбра из L в R и не использует рёбра из R в L. (На рисунке $f_e = c_e$ и $f_{e'} = 0$.) Таким образом, поток в сети (который можно контролировать на «таможенной границе» между L и R) в точности равен пропускной способности разреза. Теорема Форда—Фалкерсона доказана — если принять на веру, что алгоритм закончит работу.

7.2.5. Время работы

Каждая итерация нашего алгоритма поиска максимального потока требует $O(|E|)$ шагов, если для поиска пути из s в t использовать поиск в глубину или в ширину. Но сколько всего делается итераций? (И вообще, будет ли это число конечным?)

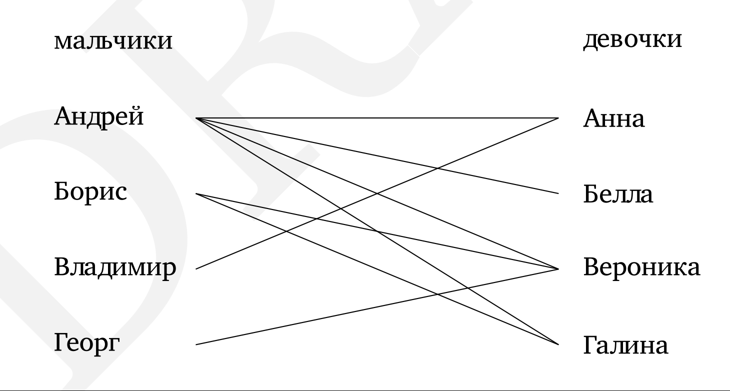
Завершение алгоритма очевидно, если пропускные способности *целые*. Тогда все числа будут целыми, на каждом шагу поток увеличивается хотя бы на 1, и всего шагов не больше $C|E|$, где C — максимальная пропускная способность ребра (почему?). Но это не очень хорошая оценка: а что если C порядка миллионов? И что будет с нецелыми (или, хуже того, иррациональными) рёбрами?

Мы ещё вернёмся к этой проблеме в упражнении 7.31. Оказывается, можно построить плохие примеры, в которых количество итераций пропорционально C, *если* неудачно выбирать пути из s в t. Но если пути выбраны разумно — например, при помощи поиска в ширину, который ищет путь с наименьшим числом рёбер, — то можно доказать, что количество итераций не больше $O(|V| \cdot |E|)$, какими бы ни были пропускные способности. Последняя оценка даёт общее время работы $O(|V| \cdot |E|^2)$ для алгоритма поиска максимального потока.

7.3. Паросочетания в двудольных графах

На рисунке 7.7 в виде графа изображены симпатии (обоюдные) между мальчиками (вершины слева) и девочками (справа)[1]. Можно ли разбить (скажем, для вальса) всех участников на пары симпатичных друг другу? В терминах теории графов: существует ли *совершенное паросочетание* (perfect matching) в этом двудольном графе?

Рис. 7.7. Рёбрами соединены ребята, которые нравятся друг другу. Можно ли разбить всех на пары по симпатиям?

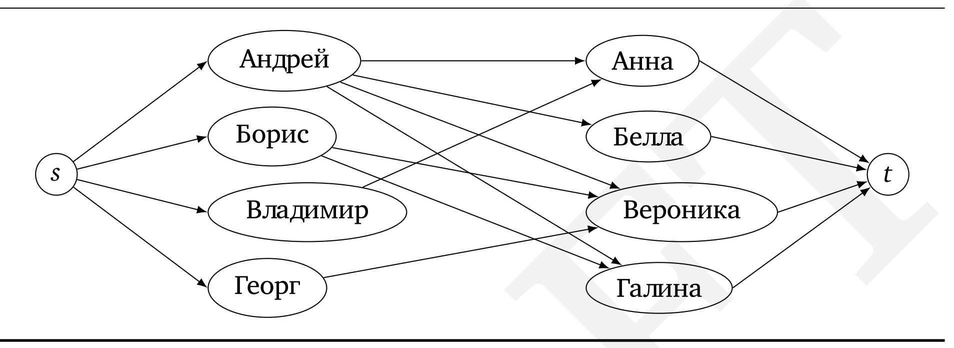


Этот вопрос можно свести к поиску максимального потока (и тем самым к задаче линейного программирования)! Создадим новую вершину-исток s с исходящими рёбрами во всех мальчиков и новую вершину-сток t со входящими рёбрами из всех девочек. Направим все рёбра в исходном двудольном

[1]Такой граф, в котором вершины поделены на две группы и рёбра проходят *между* группами, называется *двудольным* (bipartite).

графе слева направо (рис. 7.8) и присвоим каждому ребру пропускную способность 1. Совершенное паросочетание существует тогда и только тогда, когда в этой сети имеется поток, размер которого равен числу пар. Понятно ли, почему? Можете ли вы найти такой поток в нашем примере?

Рис. 7.8. Сеть для поиска паросочетания. Пропускная способность каждого ребра равна единице.



Заметили ли вы тонкий момент в сказанном? Паросочетанию соответствует максимальный поток, но в обратную сторону нужен не просто поток, а *целочисленный* поток. (Вряд ли Анна может танцевать с Андреем на 70% и с Владимиром на 30%!)

К счастью, *если все пропускные способности рёбер целочисленные, то найденный нашим алгоритмом оптимальный поток будет целочисленным.* В самом деле, если все рёбра имеют целочисленную пропускную способность, то нецелым числам неоткуда взяться.

Это верно для любой сети: если пропускные способности рёбер целые, то оптимальное решение можно искать среди целочисленных (и наш алгоритм это делает). К сожалению, для задач линейного программирования это скорее исключение, чем правило: часто бывает, что для задачи с целочисленными ограничениями оптимум достигается только на нецелочисленных решениях. А если мы ограничимся только целочисленными решениями, то (как мы увидим в главе 8) поиск оптимального среди них — и даже *хоть какого-нибудь* целочисленного решения — становится очень сложной задачей.

7.4. Принцип двойственности

Мы видели, что в сетях любой поток не превосходит любого разреза, и есть совпадающая пара (величина потока равна пропускной способности разреза). Найдя такую пару, мы тем самым убеждаемся, что в сети нет ни большего потока, ни меньшего разреза. Оказывается, что подобная удивительная вещь происходит вообще со всеми задачами линейного программирования: любая задача максимизации имеет *двойственную* задачу минимизации, и они связаны друг с другом примерно так же, как потоки и разрезы.

Чтобы понять, что это за двойственность, вспомним нашу первую линейную программу с двумя сортами шоколада:

$$\begin{aligned} x_1 + 6x_2 &\to \max \\ x_1 &\leqslant 200 \\ x_2 &\leqslant 300 \\ x_1 + x_2 &\leqslant 400 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

Симплекс-метод находит оптимальную точку $(x_1, x_2) = (100, 300)$ со значением целевой функции 1900. Сразу же ясно, что этот вариант не так далёк от оптимума: сложив два первых неравенства (второе — с коэффициентом 6), мы получим

$$x_1 + 6x_2 \leqslant 2000,$$

так что выручка больше 2000 не получится. Можно ли сложить ограничения с другими коэффициентами (добавив пока не использованные, если надо) и получить лучшую оценку? Методом проб и ошибок можно найти, что сложение второго и третьего неравенств с коэффициентами 5 и 1 соответственно даёт

$$x_1 + 6x_2 \leqslant 1900.$$

Таким образом, 1900 действительно оптимум! Этот магический приём *подтверждения оптимальности* мы уже обсуждали — но по-прежнему неясно, всегда ли такое возможно, и если возможно, то как искать коэффициенты?

Чтобы ответить на этот вопрос, представим себе, что мы ещё не знаем этих коэффициентов, и обозначим эти неизвестные коэффициенты (при первых трёх неравенствах) через y_1, y_2, y_3:

Множитель	Неравенство
y_1	$x_1 \leqslant 200$
y_2	$x_2 \leqslant 300$
y_3	$x_1 + x_2 \leqslant 400$

Во-первых, эти y_i должны быть неотрицательными, иначе неравенство поменяло бы знак. Если это так, то мы получаем оценку

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leqslant 200y_1 + 300y_2 + 400y_3.$$

Во-вторых, мы хотим, чтобы левая часть совпала бы с целевой функцией $x_1 + 6x_2$, и тогда правая часть будет верхней оценкой для оптимального решения. Другими словами, нужно получить $y_1 + y_3 = 1$ и $y_2 + y_3 = 6$. Достаточно даже $y_1 + y_3 \geqslant 1$ и $y_2 + y_3 \geqslant 6$, поскольку тогда мы получим более сильное

неравенство.[1] В итоге мы имеем оценку сверху:

$$x_1 + 6x_2 \leqslant 200y_1 + 300y_2 + 400y_3 \quad \text{при} \quad \begin{cases} y_1, y_2, y_3 \geqslant 0 \\ y_1 + y_3 \geqslant 1 \\ y_2 + y_3 \geqslant 6 \end{cases}$$

Как найти подходящие y_i? Можно попросту взять их достаточно большими, скажем, положить $(y_1, y_2, y_3) = (5, 3, 6)$. Но тогда оценка получится плохой (в нашем случае $200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4300$). Чтобы оценка была как можно более точной, мы должны минимизировать $200y_1 + 300y_2 + 400y_3$ при соблюдении указанных выше неравенств. *Снова приходим к задаче линейного программирования*:

$$\begin{gathered} 200y_1 + 300y_2 + 400y_3 \to \min \\ y_1 + y_3 \geqslant 1 \\ y_2 + y_3 \geqslant 6 \\ y_1, y_2, y_3 \geqslant 0 \end{gathered}$$

Из построения следует, что всякое решение этой *двойственной* задачи даёт некоторую оценку сверху для исходной (*прямой*) задачи. Так что если мы каким-то чудом нашли пару решений для той и другой задачи с совпадающими значениям, то оба они будут оптимальными. Вот такая пара для задачи про шоколад:

прямая: $(x_1, x_2) = (100, 300)$; двойственная: $(y_1, y_2, y_3) = (0, 5, 1)$.

В обоих случаях получается 1900, так что оба решения оптимальны. Символически это изображено на рис. 7.9.

Рис. 7.9. Все значения для прямой задачи не превосходят значений для двойственной, так что первое множество лежит слева от второго. Вопрос только в том, есть ли между ними зазор — и теорема о двойственности говорит, что нет.

оптимум прямой задачи
оптимум двойственной задачи
значение целевой функции
значения прямой задачи
значения двойственной задачи
зазор (на самом деле нулевой)

На самом деле это не чудо, а закономерность — подобную вещь можно проделать для любой задачи линейного программирования. Для удобства будем считать, что в ней используются только неотрицательные переменные,

[1]Это соответствует добавлению остававшихся без дела неравенств $-y_1 \leqslant 0$ и $-y_2 \leqslant 0$.

а ограничения имеют вид неравенств. Вот как тогда строится двойственная задача:

- Вводим неопределённый коэффициент y_j для каждого неравенства в прямой задаче; это будут (неотрицательные) переменные двойственной задачи.
- Для каждой переменной x_i в прямой задаче пишем неравенство (ограничение на y_j): при сложении неравенств прямой задачи (с множителями y_j) при x_i должен получаться коэффициент, не меньший коэффициента в целевой функции прямой задачи.
- Целевой функцией двойственной задачи будет линейная комбинация коэффициентов y_i, получающаяся в правой части при сложении неравенств прямой задачи с этими коэффициентами.

Поначалу это описание выглядит запутанным, но в матричной записи всё выглядит совсем просто (рис. 7.10). Более общий случай, когда среди условий есть и равенства, показан на рис. 7.11. Если прямая задача имеет ограничение-равенство, то соответствующий множитель (*двойственная переменная*) не обязан быть неотрицательным: равенства можно умножать на любые числа. Всё это выглядит довольно симметрично: в матрице $\mathbf{A} = (a_{ij})$ каждому прямому ограничению соответствует *строка*, а каждому двойственному — *столбец*.

Рис. 7.10. Стандартная задача линейного программирования в матричной форме и её двойственная.

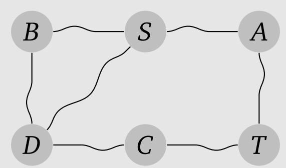
Прямая задача:	Двойственная задача:
$\mathbf{c}^T\mathbf{x} \to \max$	$\mathbf{y}^T\mathbf{b} \to \min$
$\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$	$\mathbf{y}^T\mathbf{A} \geqslant \mathbf{c}^T$
$\mathbf{x} \geqslant 0$	$\mathbf{y} \geqslant 0$

Рис. 7.11. В общем случае ограничения прямой задачи состоят из линейных неравенств (при $i \in I$) и уравнений (при $i \in E$), всего $m = |I| + |E|$ ограничений. Некоторые из n переменных x_j (при $j \in N$) обязаны быть неотрицательными. В двойственной задаче будет $m = |I| + |E|$ переменных. Те из них, которые соответствуют неравенствам, должны быть неотрицательными; остальные могут быть любыми.

Прямая задача:	Двойственная задача:
$c_1x_1 + \ldots + c_nx_n \to \max$	$b_1y_1 + \ldots + b_my_m \to \min$
$a_{i1}x_1 + \ldots + a_{in}x_n \leqslant b_i$ для $i \in I$	$a_{1j}y_1 + \ldots + a_{mj}y_m \geqslant c_j$ для $j \in N$
$a_{i1}x_1 + \ldots + a_{in}x_n = b_i$ для $i \in E$	$a_{ij}y_1 + \ldots + a_{mj}y_m = c_j$ для $j \notin N$
$x_j \geqslant 0$ для $j \in N$	$y_i \geqslant 0$ для $i \in I$

Визуализация двойственности

Задачу о кратчайшем пути в неориентированном графе можно решать с помощью *физической модели*. Свяжем вершины графа верёвками по рёбрам, причём длину верёвки возьмём равной длине соответствующего ребра. Теперь для поиска кратчайшего пути из s в t достаточно *потянуть* за s и t в разные стороны: интуиция подсказывает, что натянутым окажется кратчайший путь из s в t.



Хотя этот процесс нагляден, в нём есть одна странность: задача о кратчайшем пути вообще-то является задачей *минимизации*, а в физической модели мы *максимизируем* расстояние между s и t. Отчего такая разница? Ответ: оттягивая s от t, мы решаем задачу, *двойственную* к задаче о кратчайшем пути! Эта двойственная задача имеет очень простой вид (упражнение 7.28) и использует по одной переменной x_u на каждую вершину u:

$$x_s - x_t \to \max,$$
$$|x_u - x_v| \leqslant w_{uv} \quad \text{для всех рёбер } \{u, v\}.$$

Целевая функция соответствует желанию отдалить s от t, а ограничения соответствуют верёвкам, не дающим расстоянию между u и v превысить w_{uv}.

По построению, каждое допустимое решение двойственной задачи даёт верхнюю оценку для всякого допустимого решения прямой. Оказывается, что их оптимумы совпадают!

Теорема о двойственности. *Если линейная программа имеет ограниченный оптимум, то двойственная программа также имеет ограниченный оптимум, и они равны.*

Мы не приводим доказательство этой теоремы. Скажем только, что если прямая программа описывает задачу о максимальном потоке, то двойственную линейную программу можно интерпретировать как задачу о минимальном разрезе (упражнение 7.25). Подобно тому, как алгоритм поиска максимального потока одновременно находил и минимальный разрез (и устанавливал их равенство), симплекс-метод может найти совпадающие оптимальные значения для прямой и двойственной задач, тем самым доказав утверждение теоремы.

7.5. Игры с нулевой суммой

Различные конфликтные ситуации из реальной жизни можно представить в виде *матричных игр*. Например, всем известная игра «камень-ножницы-бу-

мага» задаётся изображённой ниже *матрицей выплат*. Игроки Строка и Столбец выбирают по элементу из множества $\{К, Н, Б\}$, после чего Столбец платит Строке сумму, равную соответствующему элементу матрицы. (Другими словами, в матрице записана прибыль Строки и убыток Столбца.)

$G =$

		Столбец		
		К	*Н*	*Б*
Строка	*К*	0	1	−1
	Н	−1	0	1
	Б	1	−1	0

Пусть теперь эти двое играют несколько раз подряд. Если Строка всё время делает один и тот же ход, то Столбец быстро подстроится и всегда будет побеждать. Чтобы избежать этого, Строка может применять *смешанную стратегию* (mixed strategy) и делать ход *К* с вероятностью x_1, ход *Н* с вероятностью x_2 и ход *Б* с вероятностью x_3. Эта стратегия задаётся вектором $\mathbf{x} = (x_1, x_2, x_3)$ из неотрицательных чисел, дающих в сумме 1. Аналогично, смешанная стратегия Столбца — это некоторый вектор $\mathbf{y} = (y_1, y_2, y_3)$.[1]

Мы считаем, что игроки действуют независимо, поэтому вероятность того, что Строка сделает ход i, а Столбец — ход j, равна $x_i y_j$. Соответственно, *ожидаемый* (средний) платёж (Столбец платит Строке) равен

$$\sum_{i,j} G_{ij} \cdot \mathrm{P}(\text{Строка делает ход } i, \text{Столбец делает ход } j) = \sum_{i,j} G_{ij} x_i y_j.$$

Строка хочет *максимизировать* эту величину, а Столбец — *минимизировать*. На что они могут рассчитывать?

Ожидаемый платёж зависит от выбранных стратегий. Пусть, например, Строка считает варианты хода равновероятными (то есть $\mathbf{x} = (1/3, 1/3, 1/3)$). Каков будет ожидаемый платёж для разных вариантов хода Столбца? Это легко подсчитать, посмотрев на матрицу игры. Если Столбец делает ход *К*, то средний платёж вычисляется по первому столбцу матрицы и равен

$$\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot (-1) + \frac{1}{3} \cdot 1 = 0.$$

Аналогичное вычисление даёт 0 и для двух других вариантов хода Столбца (*Н* или *Б*). Для смешанных стратегий (y_1, y_2, y_3) платёж является взвешенным средним платежей всех трёх вариантов *К*, *Н* и *Б*, поэтому он тоже равен нулю. Это можно увидеть и из предыдущей формулы:

$$\sum_{i,j} G_{ij} x_i y_j = \sum_{i,j} G_{ij} \cdot \frac{1}{3} y_j = \sum_j y_j \left(\sum_i \frac{1}{3} G_{ij} \right) = \sum_j y_j \cdot 0 = 0,$$

где предпоследнее равенство следует из того, что элементы каждого столбца матрицы G в сумме дают 0.

[1]Вообще говоря, игроки могут менять стратегии от раунда к раунду, но для упрощения анализа мы такую возможность не рассматриваем.

Мы видим, что если Строка следует выбранной стратегии, то ожидаемый платёж будет нулевым независимо от стратегии Столбца, так что у Строки есть гарантированный способ (в среднем) не проиграть. Для Столбца ситуация аналогична: равновероятная стратегия гарантирует ему нулевой ожидаемый платёж.

Что ж тут интересного — ведь с самого начала было ясно, что игра в «камень-ножницы-бумагу» симметрична? На самом деле мы доказали нечто большее. Представим себе, что Строка сначала объявляет свою стратегию, и Столбец знает об этом, делая свои ходы. Это, казалось бы, даёт ему преимущество по сравнению с вариантом, когда он сам объявляет свою стратегию (и Строка выбирает ответную стратегию после этого). Наш анализ показывает, тем не менее, что такого преимущества для смешанных стратегий нет.

Это удивительное свойство верно для произвольных матриц платежей: оно следует из принципа двойственности в линейном программировании (на самом деле, эквивалентно ему). В качестве примера рассмотрим несимметричную игру. Представьте себе президентские выборы, в которых имеются два кандидата на пост, и каждый из них должен выбрать главную тему предвыборной кампании из двух вариантов (буквы обозначают *экономику, общество, нравственность, снижение налогов*). Элементы платёжной матрицы — это миллионы голосов, теряемые Столбцом. (Мы предполагаем, что выбор темы осуществляется до начала кампании и изменён быть не может.)

$G =$

	Н	*С*
Э	3	−1
О	−2	1

Пусть Строка объявляет, что она будет использовать смешанную стратегию $\mathbf{x} = (1/2, 1/2)$. Что может сделать Столбец? Ход *Н* приведёт к ожидаемой потере $1/2$, а ход *С* приведёт к ожидаемой потере 0. Таким образом, наилучшим ответным ходом Столбца будем *чистая* стратегия $\mathbf{y} = (0, 1)$.

В общем случае, если фиксирована стратегия Строки $\mathbf{x} = (x_1, x_2)$, то всегда найдётся *чистая* стратегия, которая оптимальна для Столбца: либо делать ход *Н* с ожидаемым платежом $3x_1 - 2x_2$, либо ход *С* с платежом $-x_1 + x_2$, смотря по тому, какой меньше. Всякая смешанная стратегия $\mathbf{y}$ является взвешенным средним этих двух чистых стратегий, и поэтому не может побить лучшую из них.

Таким образом, если Строка должна объявить $\mathbf{x}$ до того, как сыграет Столбец, и рассчитывает на самый худший случай, в котором ожидаемый платёж равен $\min\{3x_1 - 2x_2, -x_1 + x_2\}$, то возникает такая оптимизационная задача:

$$\text{выбрать } (x_1, x_2)\text{, максимизирующее } \min\{3x_1 - 2x_2, -x_1 + x_2\}$$

Этот выбор вероятностей x_i даёт Строке лучшую из возможных *гарантий* ожидаемого платежа. Эту задачу можно записать в виде линейной программы, заметив, что *для фиксированных* x_1 и x_2 величину

$$z = \min\{3x_1 - 2x_2, -x_1 + x_2\}$$

можно описать как максимальное значение в такой оптимизационной задаче:

$$\begin{aligned} z &\to \max \\ z &\leqslant 3x_1 - 2x_2 \\ z &\leqslant -x_1 + x_2 \end{aligned}$$

Вспоминая, что мы должны выбрать x_1 и x_2 так, чтобы максимизировать это z, получаем такую задачу:

$$\begin{aligned} z &\to \max \\ -3x_1 + 2x_2 + z &\leqslant 0 \\ x_1 - x_2 + z &\leqslant 0 \\ x_1 + x_2 &= 1 \\ x_1, x_2 &\geqslant 0 \end{aligned}$$

Аналогично, если Столбец должен объявить первым о своей стратегии, то лучшее, что он может сделать, — это выбрать смешанную стратегию $\mathbf{y}$, минимизирующую его потери при лучшем ответе Строки. Другими словами, он должен

$$\text{выбрать } (y_1, y_2), \text{ минимизирующее } \underbrace{\max\{3y_1 - y_2, -2y_1 + y_2\}}_{\substack{\text{выигрыш Строки при наилучшем} \\ \text{для неё ответе на } \mathbf{y}}}.$$

В виде ЛП:

$$\begin{aligned} w &\to \min \\ -3y_1 + y_2 + w &\geqslant 0 \\ 2y_1 - y_2 + w &\geqslant 0 \\ y_1 + y_2 &= 1 \\ y_1, y_2 &\geqslant 0 \end{aligned}$$

Теперь ключевым наблюдением является то, что *эти две задачи являются двойственными друг для друга* (см. рисунок 7.11)! Таким образом, они имеют одинаковый оптимум, назовём его V.

Что же у нас получилось? При помощи решения некоторой задачи линейного программирования Строка (максимизатор) может найти стратегию, которая гарантирует ей ожидаемый выигрыш не менее V, что бы ни делал Столбец. А при помощи решения двойственной ЛП Столбец (минимизатор) может гарантировать ожидаемый платёж не более V, что бы ни делала Строка. Такое значение V называется *ценой* (value) игры. В нашем случае цена игры равна $1/7$, и она достигается смешанной стратегией $(3/7, 4/7)$ для Строки, а также смешанной стратегией $(2/7, 5/7)$ для Столбца.

Конкретный вид матрицы (и её размер) не играет роли — мы фактически доказали (вывели из принципа двойственности в линейном программировании) фундаментальный результат теории игр, доказанный фон Нейманом и

называемый *принципом минимакса* (min-max theorem). Его можно записать в виде следующего равенства:

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{ij} x_i y_j = \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{ij} x_i y_j.$$

Он показывает, что в случае смешанных стратегий открытое объявление используемой стратегии (в левой части это делает Строка, в правой — Столбец) не даёт противнику преимущества: принцип двойственности (как и в случае максимальных потоков и минимальных разрезов) делает обе части равными.

7.6. Симплекс-метод

Мы уже несколько раз упоминали «симплекс-метод» решения задач линейного программирования. Что же это такое?

Симплекс-метод получает набор линейных неравенств и линейную целевую функцию и находит оптимальную допустимую точку при помощи следующей стратегии:

```
v ← произвольная вершина допустимой области
пока у v есть сосед v′ с лучшим значением целевой функции:
  v ← v′
```

Слова «вершина» и «сосед» были понятны в наших двумерных и трёхмерных примерах (рис. 7.1 и 7.2). Что означают они для случая n переменных $x_1, \ldots, x_n$?

Набор значений x_i задаёт точку в векторном пространстве $\mathbb{R}^n$. Линейное уравнение с n переменными $x_1, \ldots, x_n$ задаёт *гиперплоскость*, разделяющую пространство на две части, а соответствующее линейное неравенство задаёт одну из этих частей — *полупространство* (включая саму гиперплоскость, так как неравенства нестрогие). Наконец, множество допустимых решений линейной программы задаётся набором неравенств и поэтому представляет собой пересечение соответствующих полупространств — *выпуклый многогранник*.[1]

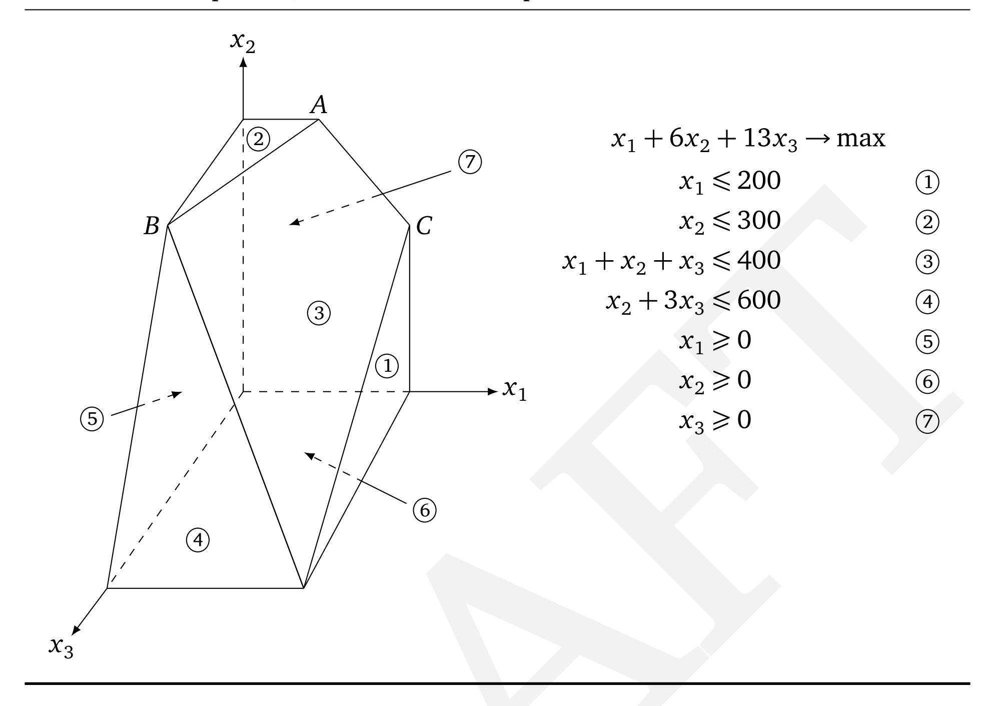
Но что в нашем общем случае означают понятия *вершина* и *сосед*?

7.6.1. Вершины и соседи в n-мерном пространстве

На рисунке 7.12 мы возвращаемся к уже встречавшемуся примеру. Если посмотреть на него внимательно, то мы видим, что *каждая вершина может быть задана как единственная точка пересечения некоторого набора гиперплоскостей*. Например, вершина A — это единственная точка, в которой ограничения ②, ③ и ⑦ обращаются в равенства. С другой стороны, гиперплоскости, соответствующие неравенствам ③ и ⑥, не задают (сами по себе, без третьей гиперплоскости) какую-либо вершину, так как их пересечение — это не одна точка, а целая прямая.

[1]Точнее было бы говорить о выпуклой многогранной области: она может быть неограниченной (уходить в бесконечность), и такие пересечения полупространств обычно не считают многогранниками.


Рис. 7.12. Многогранник, заданный семью неравенствами.



> Давайте превратим это в чёткое определение.
>
> Выберем некоторое подмножество неравенств. Если существует единственная точка, в которой они обращаются в равенства, и эта точка допустима, то она называется *вершиной*.

Сколько уравнений требуется, чтобы однозначно задать точку? Когда имеется n переменных, нам нужно по крайней мере n линейных уравнений, если требуется единственное решение. С другой стороны, если уравнений больше n, то по крайней мере одно из них является линейной комбинацией других, а поэтому может не учитываться. Подытожим:

> Каждая вершина задаётся множеством из n неравенств, обращающихся в ней в равенства.[1]

Теперь можно определить *соседство*.

> Две вершины называются *соседними*, если они имеют $n - 1$ общих определяющих неравенств.

Например, на рисунке 7.12 вершины A и C имеют общие определяющие неравенства {③,⑦}, и поэтому они соседние.

[1]Здесь есть тонкость: одна и та же вершина может порождаться различными подмножествами неравенств. На рисунке 7.12 вершина B порождается подмножеством {②,③,④}, но также и {②,④,⑤}. Такие вершины называются *вырожденными* и требуют особого подхода. Мы рассмотрим эту ситуацию позже, а пока давайте считать, что вырожденных вершин в нашей задаче нет.

7.6.2. Алгоритм

На каждой итерации симплекс-метод должен:

1) проверить, является ли текущая вершина оптимальной (и если это так, то остановиться);

2) решить, куда переходить дальше.

Как мы увидим, обе задачи просты, если текущая вершина находится в начале координат. А если вершина находится где-то ещё, то мы просто преобразуем систему координат, сделав текущую вершину началом.

Для начала посмотрим, почему начало координат удобно. Пусть мы имеем некоторую линейную программу общего вида

$$\mathbf{c}^T\mathbf{x} \to \max$$
$$\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$$
$$\mathbf{x} \geqslant 0$$

где $\mathbf{x} = (x_1, \ldots, x_n)$ — вектор из переменных. Пусть начало координат является допустимым. Тогда оно, очевидно, является вершиной, так как это единственная точка, в которой n неравенств $\{x_1 \geqslant 0, \ \ldots, \ x_n \geqslant 0\}$ обращаются в равенство. (Мы предполагаем вершину невырожденной, так что другие неравенства в равенства не обращаются.) Шаг симплекс-метода, как мы говорили, состоит из двух частей. Для первой:

> Начало координат оптимально тогда и только тогда, когда $c_i \leqslant 0$ для всех i.

Действительно, если $c_i \leqslant 0$ для всех i, то при $\mathbf{x} \geqslant 0$ рассчитывать на лучшее (положительное) значение целевой функции мы не можем. Обратно, если $c_i > 0$ для какого-то i, то начало координат не оптимально, так как мы можем увеличить значение целевой функции, увеличив x_i. (Напомним, что по предположению остальные неравенства не обращаются в равенства, так что x_i можно хотя бы немного увеличить.)

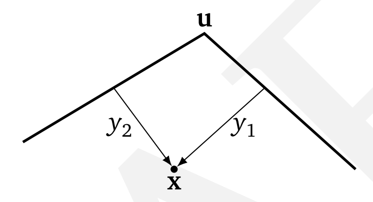
Если начало координат не оптимально, можно переместиться по некоторому ребру, увеличив некоторое x_i, для которого $c_i > 0$. Насколько мы можем увеличить x_i? *Пока мы не столкнёмся с каким-то другим ограничением.* Другими словами, мы делаем одно из определяющих вершину неравенств $x_i \geqslant 0$ строгим и увеличиваем x_i до тех пор, пока какое-то другое неравенство, до этого выполнявшееся с запасом, не обратится в равенство. При этом мы снова имеем n равенств, то есть мы находимся в новой вершине.

Например, пусть мы имеем дело со следующей линейной программой:

$$\begin{aligned} 2x_1 + 5x_2 &\to \max \\ 2x_1 - x_2 &\leqslant 4 \qquad ① \\ x_1 + 2x_2 &\leqslant 9 \qquad ② \\ -x_1 + x_2 &\leqslant 3 \qquad ③ \\ x_1 &\geqslant 0 \qquad ④ \\ x_2 &\geqslant 0 \qquad ⑤ \end{aligned}$$

Симплекс-метод можно запустить в начале координат, которое задаётся ограничениями ④ и ⑤. Для перехода мы «освобождаем» ограничение $x_2 \geqslant 0$, превращая его из равенства в строгое неравенство. Увеличивая x_2, мы прежде всего наталкиваемся на ограничение $-x_1 + x_2 \leqslant 3$, и поэтому должны остановиться на $x_2 = 3$, в точке, где это новое неравенство обращается в равенство. Новую вершину задают неравенства ③ и ④.

Таким образом, мы знаем, что делать, если мы находимся в начале координат. Но что если наша текущая вершина $\mathbf{u}$ расположена где-то ещё? Мы сводим этот случай к уже рассмотренному с помощью линейного преобразования координат. В качестве новых координат мы выбираем те самые линейные функции, которые использованы в определяющих точку неравенствах, сдвинутые в нуль. Другими словами, новые координаты точки $\mathbf{x}$ пропорциональны расстояниям $y_1, \ldots, y_n$ от точки $\mathbf{x}$ до гиперплоскостей, пересечением которых является вершина $\mathbf{u}$:



Более точно, если одно из неравенств, задающих вершину, имеет вид $\mathbf{a}_i \cdot \mathbf{x} \leqslant b_i$, то расстояние от точки $\mathbf{x}$ до *грани*, соответствующей этому неравенству, пропорционально

$$y_i = b_i - \mathbf{a}_i \cdot \mathbf{x}.$$

Получаем n линейных функций y_i, по одной на каждую грань, причём переход от $\mathbf{x}$ к $\mathbf{y}$ можно обратить и выразить x_i как линейную функцию от y_j. Таким образом, мы можем переписать всю линейную программу в переменных y_i. Получаем эквивалентную линейную программу (например, оптимальное значение остаётся тем же), но записанную в другой системе координат.

1. Новая программа содержит неравенства $\mathbf{y} \geqslant 0$, в которые преобразуются неравенства, задающие $\mathbf{u}$.

2. Точка $\mathbf{u}$ становится началом координат $\mathbf{y}$-пространства.

3. Целевой функцией становится $c_\mathbf{u} + \tilde{\mathbf{c}}^T \mathbf{y}$, где $c_\mathbf{u}$ — это значение целевой функции в $\mathbf{u}$, а $\tilde{\mathbf{c}}$ — преобразованный вектор коэффициентов целевой функции в новых координатах.

В результате мы вернулись к случаю, с которым уже умеем работать! На рисунке 7.13 описанный алгоритм применяется к нашему примеру.

Теперь симплекс-метод полностью описан. Он переходит от вершины к соседней вершине, останавливаясь, когда дальнейшее увеличение целевой функции невозможно, то есть когда все коэффициенты её выражения в новых координатах меньше или равны 0. Как мы уже говорили, вершина с этим свойством даёт глобальный оптимум. Пока текущая вершина не является локально оптимальной, её локальная система координат включает некоторое

Рис. 7.13. Пример работы симплекс-метода.

Исходная линейная программа: $2x_1 + 5x_2 \to \max$ $2x_1 - x_2 \leqslant 4$ ① $x_1 + 2x_2 \leqslant 9$ ② $-x_1 + x_2 \leqslant 3$ ③ $x_1 \geqslant 0$ ④ $x_2 \geqslant 0$ ⑤	*Текущая вершина*: {④, ⑤} (начало координат). *Значение целевой функции*: 0. *Шаг*: увеличить x_2. ⑤ перестаёт быть равенством, ③ обращается в равенство. Остановиться при $x_2 = 3$. Локальные координаты: {④, ③}: $y_1 = x_1, \quad y_2 = 3 + x_1 - x_2$
Переписанная линейная программа в координатах (y_1, y_2): $15 + 7y_1 - 5y_2 \to \max$ $y_1 + y_2 \leqslant 7$ ① $3y_1 - 2y_2 \leqslant 3$ ② $y_2 \geqslant 0$ ③ $y_1 \geqslant 0$ ④ $-y_1 + y_2 \leqslant 3$ ⑤	*Текущая вершина*: {④, ③}. *Значение целевой функции*: 15. *Шаг*: увеличить y_1. ④ перестаёт быть равенством, ② обращается в равенство. Остановиться при $y_1 = 1$. Локальные координаты: {②, ③}: $z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$
Линейная программа в координатах (z_1, z_2): $22 - \frac{7}{3}z_1 - \frac{1}{3}z_2 \to \max$ $-\frac{1}{3}z_1 + \frac{5}{3}z_2 \leqslant 6$ ① $z_1 \geqslant 0$ ② $z_2 \geqslant 0$ ③ $\frac{1}{3}z_1 - \frac{2}{3}z_2 \leqslant 1$ ④ $\frac{1}{3}z_1 + \frac{1}{3}z_2 \leqslant 4$ ⑤	*Текущая вершина*: {②, ③}. *Значение целевой функции*: 22. *Текущая вершина оптимальна*: все $c_i < 0$. Ищем координаты точки {②, ③} (в исходной системе координат). Получаем оптимальное решение $(x_1, x_2) = (1, 4)$.

{②, ③}
увеличить y_1
{③, ④}
{①, ②}
увеличить x_2
{④, ⑤}
{①, ⑤}

направление, по которому целевую функцию можно улучшить, так что мы сдвигаемся в этом направлении (по соответствующему ребру многогранника), пока не попадём в соседнюю вершину. В силу предположения о невырожденности (см. сноску на с. 210), это ребро имеет ненулевую длину, и поэтому мы улучшаем значение целевой функции. Таким образом, процесс рано или поздно должен остановиться (вершин конечное число).

7.6.3. Тонкости

В симплекс-методе есть ряд важных моментов, которые мы ещё не рассмотрели.

Начальная вершина. Как нам найти вершину, из которой запускать алгоритм? В наших двумерных и трёхмерных примерах мы всегда начинали из начала координат: эта точка не только лежала в области допустимых значений, но и была её вершиной. Для произвольной линейной программы это не гарантировано: вообще говоря, мы заранее не знаем ни одной допустимой точки. Но оказывается, поиск такой точки *можно свести к некоторой линейной программе,* которую можно решить с помощью того же симплекс-метода!

Чтобы увидеть, как это делается, начнём с произвольной линейной программы в стандартном виде (см. раздел 7.1.4):

$$\mathbf{c}^T\mathbf{x} \to \min, \quad \text{где } \mathbf{A}\mathbf{x} = \mathbf{b} \text{ и } \mathbf{x} \geqslant 0.$$

(Мы хотим найти какое-либо решение; на этом этапе вектор **c** роли не играет.) Можно считать, что все правые части уравнений неотрицательны: если $b_i < 0$, то мы просто умножаем обе части i-го уравнения на -1.

Далее мы следующим образом записываем новую линейную программу:

- Вводим m *вспомогательных переменных* $z_1, \ldots, z_m \geqslant 0$, где m — количество уравнений.
- Модифицируем уравнения, добавляя z_i к левой части i-го уравнения.
- Целевой функцией для *минимизации* будет $z_1 + z_2 + \ldots + z_m$.

Для этой новой линейной программы легко получить допустимую точку — пусть $z_i = b_i$ для всех i, а все другие переменные равны нулю. Затем можно воспользоваться симплекс-методом, чтобы получить оптимальное решение.

Имеются две возможности. Если в этом решении значение $z_1 + \ldots + z_m$ нулевое, то все z_i — нули, а значит, оптимальная вершина для новой линейной программы является допустимой вершиной исходной линейной программы. Мы можем забыть про z_i и (наконец-то) запустить симплекс-метод для исходной задачи![1]

Но оптимальное значение целевой функции может оказаться ненулевым. Что это означает? Мы пытались минимизировать сумму z_i, но симплекс-метод установил, что она не может быть нулевой. Значит, исходная линейная программа несовместна: ей требуются какие-то ненулевые z_i, чтобы свести концы с концами. Тем самым мы, использовав симплекс-метод, обнаружили несовместность линейной программы.

Вырождение. У многогранника на рисунке 7.12 вершина *B вырожденная*, то есть принадлежит более чем трём граням многогранника (в данном случае

[1]Тут есть ещё одна тонкость: имея допустимую точку, надо найти вершину. Изменяя значение какой-то из переменных до тех пор, пока одно из неравенств не превратится в равенство, можно попасть на границу области. Затем можно двигаться по грани, сохраняя полученное равенство, пока ещё одно неравенство не обратится в равенство, и так далее. Мы не будем обсуждать это подробно.

②, ③, ④, ⑤). Алгебраически это означает, что из четырёх плоскостей, проходящих через эту вершину, можно оставить любые три ({②,③,④}, {②,③,⑤}, {②,④,⑤} или {③,④,⑤}), и соответствующая система с тремя уравнениями от трёх неизвестных имеет единственное решение — нашу вершину (0, 300, 100).

Для симплекс-метода это создаёт проблемы: естественно задавать вершину набором неравенств, которые в ней обращаются в равенства, а соседство понимать как замену одного из равенств на другое. Но в вырожденном случае такое задание уже не однозначно и соседство уже так просто не описывается. Другая возможность вырождения — когда движение по ребру не меняет значения целевой функции (что создаёт потенциальную возможность зацикливания).

Один из способов избавиться от этих проблем — произвести *шевеление* (perturbation): добавить к коэффициентам небольшие случайно выбранные числа. С практической точки зрения это не играет роли, так как изменения очень малы. Зато теперь можно быть уверенным, что вырожденных вершин не будет — вероятность того, что после случайного шевеления сдвинутые плоскости ②, ③, ④, ⑤ по-прежнему пересекутся в одной точке, равна нулю.

Неограниченность. В некоторых случаях линейная программа не ограничена в том смысле, что целевая функция может быть сделана произвольно большой (или малой, если речь о задаче минимизации). В таком случае симплекс-метод это обнаружит следующим образом. На каждой итерации алгоритм пытается увеличить текущее значение целевой функции, увеличивая некоторое x_i, для которого соответствующий коэффициент c_i целевой функции строго положителен. При этом то, насколько можно увеличить x_i, определяется оставшимися ограничениями. Если же эти ограничения не мешают увеличивать x_i до бесконечности, то целевая функция не ограничена. В этом случае симплекс-метод останавливается и сообщает об ошибке.

7.6.4. Время работы симплекс-метода

Каково время работы симплекс-метода для линейной программы общего вида

$$\mathbf{c}^T\mathbf{x} \to \max, \quad \text{где } \mathbf{Ax} \leqslant 0 \text{ и } \mathbf{x} \geqslant 0,$$

если в ней n переменных и $\mathbf{A}$ содержит m ограничений-неравенств? Так как это итеративный алгоритм, который переходит от вершины к вершине, давайте начнём с оценки времени работы одной итерации. Пусть $\mathbf{u}$ — текущая вершина. По определению, это единственная точка, в которой n ограничений-неравенств обращаются в равенства. Каждая из соседних с ней вершин также использует $n-1$ из этих неравенств, так что $\mathbf{u}$ может иметь не более nm соседей: нужно выбрать, какое неравенство отбросить, а какое новое добавить.

Можно было бы для каждого потенциального соседа проверить, действительно ли он является некоторой вершиной многогранника, и вычислить значение целевой функции. Второе несложно (скалярное произведение требует

n умножений), но проверка того, что перед нами действительно вершина, требует решения системы из n уравнений с n неизвестными (состоящей из неравенств, обращающихся в равенства) и проверки допустимости результата. При использовании метода Гаусса (см. следующую врезку) это занимает время $O(n^3)$, что даёт большое время работы: $O(mn^4)$ на одну итерацию.

Метод Гаусса

Чтобы найти координаты вершины, надо решить систему линейных уравнений. Это можно сделать *методом Гаусса*. Пусть даны n линейных уравнений с n неизвестными, скажем такие ($n=4$):

$$\begin{aligned} x_1 \quad - 2x_3 \quad &= 2 \\ x_2 + x_3 \quad &= 3 \\ x_1 + x_2 \quad - x_4 &= 4 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

Если прибавить к одному уравнению другое (возможно, с каким-то коэффициентом), получим эквивалентную систему. Прибавим первое уравнение (умноженное на -1) к третьему:

$$\begin{aligned} x_1 \quad - 2x_3 \quad &= 2 \\ x_2 + x_3 \quad &= 3 \\ x_2 + 2x_3 - x_4 &= 2 \\ x_2 + 3x_3 + x_4 &= 5 \end{aligned}$$

Коэффициент подобран так, чтобы *исключить* переменную x_1 из третьего уравнения (оставив только одно уравнение с x_1). Получилась система с *тремя* уравнениями и *тремя* неизвестными x_2, x_3, x_4: решив её, мы подставим решение в первое уравнение и найдём x_1.

```
функция GAUSS(E, X)
{Вход: система E из n уравнений с n неизвестными X = {x_1, ..., x_n},
  i-е уравнение e_i:   a_i1 x_1 + a_i2 x_2 + ... + a_in x_n = b_i .}
{Выход: решение системы, если оно существует и единственно.}
если все коэффициенты a_i1 равны нулю:
  вернуть «система несовместна или линейно зависима»
если n = 1:  вернуть b_1/a_11
выбрать коэффициент a_p1 с наибольшим абсолютным значением
поменять местами уравнения e_1 и e_p
для i от 2 до n:
  e_i ← e_i − (a_i1/a_11) · e_1
(x_2, ..., x_n) ← GAUSS(E − {e_1}, X − {x_1})
x_1 ← (b_1 − Σ_{j>1} a_1j x_j)/a_11
вернуть (x_1, ..., x_n)
```

(Мы заботимся о том, чтобы коэффициент, на который нужно делить, был побольше — деление на малые числа приводит к большой погрешности.)

Метод Гаусса выполняет $O(n^2)$ *арифметических операций*, уменьшая n на 1; всего будет $O(n^3)$ операций. Но операции эти проводятся с рациональными числами, числители и знаменатели которых могут расти по ходу дела, так что для честной оценки времени работы надо бы ещё оценить битовый размер возникающих чисел. Это можно сделать, но в такие подробности мы входить не будем.

К счастью, можно уменьшить множитель mn^4 до mn, тем самым сделав симплекс-метод применимым на практике. Вспомним, что мы для каждой вершины переписывали систему в локальных координатах (раздел 7.6.2) и что на каждом шаге меняется только одна из этих координат. В результате на каждой итерации расходы по переписыванию линейной программы в новых локальных координатах составляют всего лишь $O((m+n)n)$.

Выбирая соседа с лучшим значением целевой функции, вспомним, что целевая функция имеет (в локальных координатах) вид $c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$, где $c_{\mathbf{u}}$ — значение целевой функции в $\mathbf{u}$. Это сразу определяет перспективное направление перехода: мы выбираем произвольное $\tilde{c}_i > 0$ (если такого нет, то текущая вершина оптимальная, и симплекс-метод останавливается). Определить, на сколько можно увеличить y_i, тоже легко, так как неравенства уже переписаны в локальных координатах, и остаётся разрешить их относительно y_i. Так мы либо находим следующую точку, либо (если можем увеличивать y_i бесконечно) узнаём, что линейная программа не ограничена.

Итак, время работы одной итерации симплекс-метода — всего $O(mn)$. Но сколько всего может быть итераций? Ясно, что их не может быть больше, чем вершин, которых не больше C_{m+n}^n. Но эта верхняя оценка экспоненциальна по n. Существуют примеры линейных программ, для которых симплекс-метод действительно производит экспоненциальное число итераций. Другими словами, *симплекс-метод является экспоненциальным по времени*. Однако подобные экспоненциальные примеры практически не встречаются, и это делает симплекс-метод полезным на практике алгоритмом.

7.7. Эпилог: вычисление значения схемы

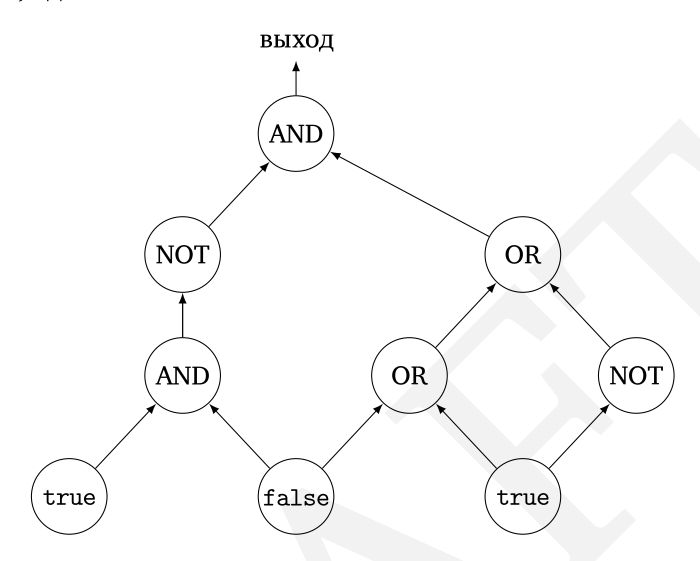
Мы уже видели, что многие задачи сводятся к задаче линейного программирования. Сейчас мы подтвердим это, показав, что в некотором смысле все задачи, разрешимые за полиномиальное время, сводятся к задаче линейного программирования.

Рассмотрим для начала одну из таких (разрешимых за полиномиальное время) задач. Пусть нам дана *булева схема*, то есть ациклический ориентированный граф, состоящий из функциональных элементов следующих типов.

- *Входные элементы* имеют нулевую входящую степень и значение `true` или `false`.

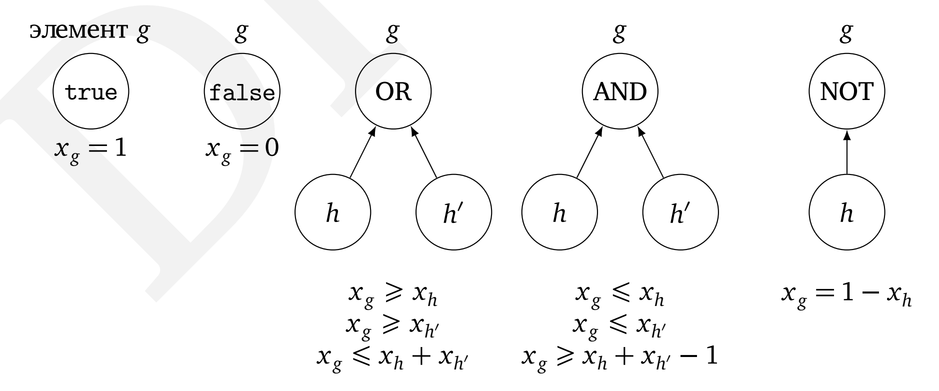
- Элементы AND и OR имеют входящую степень 2.
- Элемент NOT имеет входящую степень 1.

Кроме того, один из элементов обозначен как *выход*.



Пример булевой схемы приведён на рисунке. Задача состоит в вычислении значения схемы: что будет на выходе при данных значениях на входах? (При вычислении надо обходить элементы в порядке топологической сортировки и выполнять указанные булевы операции для уже известных входов.)

Есть очень простой способ свести эту задачу к линейному программированию.[1] Заведём переменную x_g для каждого элемента g, наложив ограничения $0 \leqslant x_g \leqslant 1$. Добавим дополнительные ограничения для элементов каждого типа:



[1]Это утверждение выглядит бессодержательным, поскольку эта задача сама разрешима за полиномиальное время, так что и линейное программирование не нужно. На самом деле в этом разделе речь идёт о более ограниченном виде сводимости, когда память сводящего алгоритма логарифмически ограничена. Мы не будем обсуждать это подробно, ограничившись описанием конкретного сведе́ния.

Линейное программирование за полиномиальное время

Симплекс-метод не является полиномиальным (время работы не ограничено полиномом от размера входа). Для некоторых линейных программ он обходит экспоненциально много вершин, хотя на практике такое случается редко. Долгое время линейное программирование считалось парадоксальной задачей, которая решается быстро на практике, но не в теории.

Однако в 1979 году молодой советский математик Леонид Хачиян доказал, что некоторый другой алгоритм решения задач линейного программирования, называемый *методом эллипсоидов*, решает любую задачу линейного программирования за полиномиальное время. Вместо того чтобы искать решение, переходя от одного угла многогранника к другому, алгоритм Хачияна заключает решение во всё меньшие эллипсоиды (сплющенные многомерные шары). Сам по себе метод довольно прост, но доказательство его корректности сложное. Публикация этого алгоритма, как и (много раньше) запуск первого спутника, стала эффектным достижением, в разгар холодной войны обеспокоившим американские власти возможным отставанием от СССР. На практике метод эллипсоидов так и не заменил симплекс-метод, несмотря на своё теоретическое значение. Вышло ещё более парадоксально: для задачи линейного программирования есть два эффективных алгоритма, но один эффективен только теоретически, а другой — только практически.

Несколько лет спустя Нарендра Кармаркар (Narendra Karmarkar), аспирант университета Беркли, придумал совершенно другую идею, которая привела к ещё одному (доказуемо) полиномиальному алгоритму для задачи линейного программирования. Алгоритм Кармаркара известен как *метод внутренней точки*, потому что он движется к оптимуму не от одной вершины к другой (по рёбрам), как это делает симплекс-метод, а разумно «срезая» путь внутри многогранника. И он таки практически полезен.

Но, возможно, наибольшим достижением оказались не теоретический прорыв Хачияна и не новый подход Кармаркара сами по себе, а возникшее соревнование между симплекс-методом и методом внутренней точки, в результате которого появились очень быстрые алгоритмы для задач линейного программирования.

Эти ограничения вынуждают все элементы принимать правильные значения — 0 для `false` и 1 для `true`. Нам не нужно ничего максимизировать или минимизировать, и мы можем прочитать ответ в переменной x_{out}, соответствующей выходному элементу.

В чём интерес задачи о вычислении значения схемы? Она в некотором смысле является *наиболее общей задачей среди всех решаемых за полиномиальное время*! Это можно объяснить примерно так: в конечном счёте всякий алгоритм исполняется компьютером, а компьютер по существу представляет собой булеву схему. Правда, один и тот же бит в памяти компьютера последовательно принимает разные значения. С учётом этого работу компьютера

можно представить в виде булевой схемы, состоящей из нескольких копий схемы компьютера, по одной на единицу времени, так что значения элементов одного уровня становятся входами следующего уровня. Если время работы алгоритма полиномиально, то и моделирующая его схема имеет полиномиальный размер. Таким образом, сводимость задачи вычисления значения схемы к линейному программированию говорит о том, что *все задачи, которые можно решить за полиномиальное время, также сводятся к линейному программированию!*

В следующих разделах, говоря о NP-полноте, мы увидим, что многие *сложные* задачи аналогичным образом сводятся к *целочисленному программированию* — задаче, которая на вид очень похожа на линейное программирование (только все переменные целые), но гораздо сложнее.

Ещё один вопрос напоследок: что ещё напоминает нам задача вычисления булевой схемы? Схема — это ориентированный ациклический граф. Какой ещё алгоритмический подход больше всего подходит для решения задач с ориентированными ациклическими графами? Динамическое программирование, конечно. Не забывайте про два этих важных метода!

Упражнения

7.1. Рассмотрим следующую линейную программу:

$$\begin{aligned} 5x + 3y &\to \max \\ 5x - 2y &\geqslant 0 \\ x + y &\leqslant 7 \\ x &\leqslant 5 \\ x &\geqslant 0 \\ y &\geqslant 0 \end{aligned}$$

Нарисуйте допустимую область и определите оптимальное решение.

7.2. Лебеду выращивают в Заплатове и Дырявине, а едят в Разутове и Неурожайке. Заплатово производит 15 возов лебеды, а Дырявино — 8. В Разутове съедают 10 возов, в Неурожайке — 13. Доставка одного воза стоит 4 алтына из Дырявина в Разутово, 1 алтын из Дырявина в Неурожайку, 2 алтына из Заплатова в Разутово и 3 алтына из Заплатова в Неурожайку.

Напишите линейную программу, которая определяет план поставок (число возов не обязано быть целым) с минимальной суммарной стоимостью доставки.

7.3. Грузовой самолёт может перевозить не более 100 тонн груза объёмом не более 60 кубических метров. Для перевозки есть три материала.

- Материал 1 имеет плотность 2 тонны на кубический метр, всего доступно 40 кубических метров, доход — 1000 руб. за кубический метр.
- Материал 2 имеет плотность в 1 тонну на кубический метр, всего доступно 30 кубических метров, доход — 1200 руб. за кубический метр.

- Материал 3 имеет плотность 3 тонны на кубический метр, всего доступно 20 кубических метров, доход — 12000 руб. за кубический метр.

Напишите линейную программу, которая оптимизирует доход при имеющихся ограничениях.

7.4. Буфетчик решает, сколько «классического» и «крепкого» пива закупать в месяц. Классическое пиво обходится ему по десять рублей за литр, а продаёт он его по двадцать рублей за литр. Крепкое пиво обходится ему в пятнадцать рублей за литр, а продаётся по тридцать. Однако поставщик капризничает и продаёт литр крепкого только вместе с двумя литрами классического в нагрузку, и к тому же почему-то отказывается продавать больше 3000 литров (всего) в месяц. Как действовать буфетчику с наибольшей выгодой? (Раскупят в любом случае всё.) Составьте линейную программу и решите её геометрически.

7.5. Компания «Белый клык» предлагает два вида еды для собак: «Борзой щенок» и «Гончие псы», сделанные из смеси каши и мяса. На пакет «Борзого щенка» уходит 1 килограмм каши и 1,5 килограмма мяса; он продаётся за 70 рублей. Пакет «Гончих псов» использует 2 килограмма каши и 1 килограмм мяса и продаётся за 60 рублей. Каша стоит 10 рублей за килограмм, а мясо стоит 20 рублей за килограмм. Производственные расходы: 14 рублей на пакет «Борзого щенка» и 6 рублей на пакет «Гончих псов». Всего в месяц можно получить до 240 тонн каши и 180 тонн мяса. Покупают всё произведённое. На производстве есть «узкое место» — за месяц фабрика может упаковать только 110000 пачек «Борзого щенка». Естественно, руководство желает максимизировать доход.

(a) Сформулируйте задачу в виде линейной программы с двумя переменными.

(b) Изобразите множество допустимых решений, запишите координаты каждой вершины и отметьте вершину с максимумом дохода. Чему равен этот максимум?

7.6. Приведите пример линейной программы с двумя переменными, у которой область допустимых значений бесконечна, но имеется оптимальное решение (с конечным значением целевой функции).

7.7. Укажите все пары вещественных чисел a, b, при которых линейная программа

$$\begin{aligned} x + y &\to \max \\ ax + by &\leqslant 1 \\ x, y &\geqslant 0 \end{aligned}$$

(a) несовместна;

(b) не ограничена;

(c) имеет единственное оптимальное решение.

7.8. Даны следующие точки на плоскости:

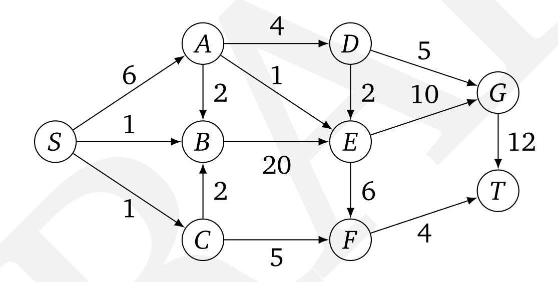
$$(1,3),\ (2,5),\ (3,7),\ (5,11),\ (7,14),\ (8,15),\ (10,19).$$

Вам нужно найти прямую $ax+by=c$, которая приблизительно проходит через эти точки (никакая прямая в точности не подойдёт). Напишите линейную программу (решать её не нужно) для поиска прямой, минимизирующей *максимальное абсолютное отклонение* (maximum absolute error)

$$\max_{1\leqslant i\leqslant 7} |ax_i+by_i-c|.$$

7.9. Задача *квадратичного программирования* состоит в максимизации квадратичной целевой функции (с членами вроде $3x_1^2$ или $5x_1x_2$) с учётом набора линейных ограничений на переменные. Приведите пример такой задачи с двумя переменными x_1, x_2, для которой допустимая область значений переменных непуста и ограничена, но никакие вершины этой области не оптимизируют (квадратичную) целевую функцию.

7.10. Для следующей сети с указанными пропускными способностями рёбер найдите максимальный поток из S в T, а также соответствующий разрез.



7.11. Рассмотрим такую задачу линейного программирования:

$$\begin{aligned} x+y &\to \max \\ 2x+y &\leqslant 3 \\ x+3y &\leqslant 5 \\ x,y &\geqslant 0 \end{aligned}$$

Какая задача будет к ней двойственной? Найдите оптимальные решения для прямой и для двойственной задач.

7.12. Для линейной программы

$$\begin{aligned} x_1-2x_3 &\to \max \\ x_1-x_2 &\leqslant 1 \\ 2x_2-x_3 &\leqslant 1 \\ x_1,x_2,x_3 &\geqslant 0 \end{aligned}$$

докажите, что решение $(x_1,x_2,x_3)=(3/2,1/2,0)$ оптимально.

7.13. *Орлянка.* В этой игре с двумя участниками один из игроков зажимает рублёвую монету в кулак орлом или решкой вверх, а второй пытается отгадать, орёл сверху или решка. Если отгадал, он забирает монету, а если нет — платит рубль штрафа.

(a) Представьте платежи в виде матрицы размера 2×2.

(b) Какова цена этой игры и каковы оптимальные стратегии для каждого из игроков?

7.14. Два конкурирующих производителя пиццы, Тони и Джонни, планируют со следующей недели разные новшества. Джонни предполагает либо понизить цены, либо увеличить порции. У Тони есть три варианта: начать выпускать пиццу лучшего качества, поставить столики на улице или предлагать газировку бесплатно. Комбинации этих вариантов приведут к таким результатам (указан выигрыш Джонни, равный убытку Тони):

		Тони		
		Качество	Столики	Газировка
Джонни	Низкие цены	+2	0	−3
	Большие порции	−1	−2	+1

Какова цена этой игры и каковы оптимальные стратегии для игроков?

7.15. Найдите цену игры, заданной следующей матрицей платежей:

$$\begin{array}{rrrr}
0 & 0 & -1 & -1 \\
0 & 1 & -2 & -1 \\
-1 & -1 & 1 & 1 \\
-1 & 0 & 0 & 1 \\
1 & -2 & 0 & -3 \\
1 & -1 & -1 & -1 \\
0 & -3 & 2 & -1 \\
0 & -2 & 1 & -1
\end{array}$$

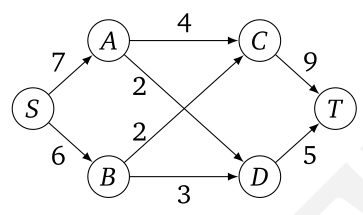
(Подсказка: рассмотрите смешанные стратегии $(1/3, 0, 0, 1/2, 1/6, 0, 0, 0)$ и $(2/3, 0, 0, 1/3)$.)

7.16. В салат входят (1) помидоры, (2) салат-латук, (3) шпинат, (4) морковь, (5) масло. Каждая порция салата должна содержать (A) по крайней мере 15 г белков, (B) не менее 2 г и не более 6 г жиров, (C) по крайней мере 4 г углеводов, (D) не более 100 мг натрия. Содержание этих веществ (в расчёте на 100 г продукта) приведено в таблице. Кроме того, (E) ваш салат не должен содержать больше 50% зелени (салата-латука и шпината) от общей массы.

ингредиент	энерг. ценность (Ккал)	белки (г)	жиры (г)	углеводы (г)	натрий (мг)
помидоры	21	0,85	0,33	4,64	9,00
салат-латук	16	1,62	0,20	2,37	8,00
шпинат	371	12,78	1,58	74,69	7,00
морковь	346	8,39	1,39	80,70	508,20
масло	884	0,00	100,00	0,00	0,00

Найдите в интернете программу для решения задач линейного программирования и используйте её, чтобы найти вариант с наименьшим количеством калорий при заданных ограничениях. Объясните, как вы составляли линейную программу и каково оптимальное решение (количество каждого ингредиента и значение калорийности). Укажите использованные источники.

7.17. Рассмотрите следующую сеть (числа обозначают пропускные способности рёбер).



(a) Найдите максимальный поток f и минимальный разрез.

(b) Изобразите остаточную сеть G^f (вместе с пропускными способностями его рёбер). На этой остаточной сети отметьте вершины, достижимые из S, и вершины, из которых достижима T.

(c) Ребро сети называется *узким местом* (bottleneck edge), если увеличение его пропускной способности приводит к увеличению максимального потока. Перечислите все узкие места для сети выше.

(d) Приведите простой пример (не больше четырёх вершин) сети без узких мест.

(e) Приведите эффективный алгоритм поиска всех узких мест в сети. (Подсказка: начните с запуска обычного алгоритма поиска максимального потока, а затем исследуйте остаточную сеть.)

7.18. Известно много вариаций задачи о максимальном потоке. Вот четыре из них.

(a) Имеется несколько истоков и несколько стоков, и нам нужно максимизировать общий поток из всех истоков во все стоки.

(b) Каждая *вершина* также имеет пропускную способность — максимальный поток, который может в неё входить.

(c) Каждое ребро имеет не только пропускную способность, но и *нижнюю оценку* на проходящий через него поток.

(d) Исходящий поток из каждой вершины u не совпадает с входящим, а получается из него умножением на $(1 - \varepsilon_u)$, где ε_u — коэффициент потерь для вершины u.

Укажите эффективные алгоритмы для этих задач, сведя (a) и (b) к обычной задаче о максимальном потоке, а (c) и (d) — к задаче линейного программирования.

7.19. Пусть кто-то утверждает, что нашёл максимальный поток в некоторой сети. Постройте *линейный* по времени алгоритм, проверяющий, действительно ли данный поток максимален.

7.20. Рассмотрим следующее обобщение задачи о максимальном потоке.

Дана ориентированная сеть $G=(V, E)$ с пропускными способностями рёбер $\{c_e\}$. Вместо одного истока и одного стока даны несколько пар (s_1, t_1), (s_2, t_2), …, (s_k, t_k), где s_i — это истоки, а t_i — стоки. Также дано k чисел $d_1, \ldots$ $\ldots, d_k$. Нужно найти k потоков $f^{(1)}, \ldots, f^{(k)}$ со следующими свойствами:

- $f^{(i)}$ является потоком из s_i в t_i.
- Для каждого ребра e суммарный поток $f_e^{(1)}+f_e^{(2)}+\ldots+f_e^{(k)}$ не превосходит пропускную способность c_e.
- Величина каждого потока $f^{(i)}$ не меньше d_i.
- Величина *общего* потока (суммы потоков) максимальна.

Как бы вы решали эту задачу?

7.21. Ребро в сети называется *критическим*, если уменьшение его пропускной способности приводит к уменьшению максимального потока. Приведите эффективный алгоритм, который находит в сети какое-нибудь критическое ребро.

7.22. В сети $G=(V, E)$, рёбра которой имеют целочисленные пропускные способности c_e, мы уже нашли максимальный поток f из вершины s в вершину t. Однако только что мы узнали, что одно из использованных значений пропускной способности было неверным: для ребра (u, v) мы использовали c_{uv}, хотя надо было $c_{uv}-1$. Это досадно, потому что поток f использовал именно это ребро на полную мощность: $f_{uv}=c_{uv}$.

Можно было бы повторить заново всё вычисление потока, но есть более быстрый метод. Покажите, как новый оптимальный поток можно вычислить за время $O(|V|+|E|)$.

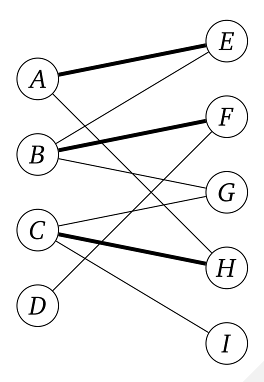
7.23. *Вершинным покрытием* неориентированного графа $G=(V, E)$ называется множество его вершин, которое покрывает каждое ребро, то есть подмножество $S\subseteq V$ с таким свойством: для каждого ребра $\{u, v\}\in E$ хотя бы один из его концов u, v лежит в S.

Покажите, что задача нахождения минимального вершинного покрытия в *двудольном* графе сводится к задаче о максимальном потоке. (Подсказка: можете ли вы как-то связать минимальное вершинное покрытие с минимальным разрезом в некоторой сети?)

7.24. *Алгоритм для двудольного паросочетания без сведения к потоку.* Мы уже знаем, как искать максимальное паросочетание в двудольном графе при помощи сведения к задаче о максимальном потоке. Теперь мы разработаем прямой алгоритм.

Пусть $G=(V_1\cup V_2, E)$ — двудольный граф (рёбра идут между V_1 и V_2), и пусть $M\subseteq E$ — некоторое паросочетание в G (то есть множество рёбер, не имеющих общих концов). Вершина называется *покрытой* паросочетанием M, если она является концом одного из рёбер M. *Чередующимся путём* (alternating path) называется путь нечётной длины, который начинается и заканчивается в непокрытых вершинах и рёбра которого поочерёдно принадлежат M и $E-M$.

(a) В изображённом ниже двудольном графе паросочетание M показано жирными линиями. Найдите чередующийся путь.

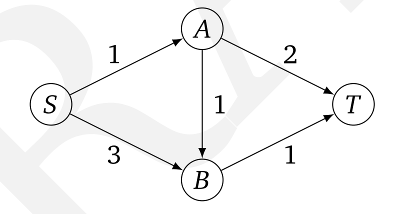


(b) Докажите, что паросочетание M максимально тогда и только тогда, когда для него не существует чередующегося пути.

(c) Разработайте алгоритм, который ищет чередующийся путь за время $O(|V|+|E|)$, используя идею поиска в ширину.

(d) Приведите прямой алгоритм поиска максимального паросочетания в двудольном графе, работающий за время $O(|V|\cdot|E|)$.

7.25. *Задача о максимальном потоке и двойственная к ней.* Рассмотрим сеть, изображённую на рисунке.



(a) Запишите задачу поиска максимального потока из S в T в виде линейной программы.

(b) Запишите двойственную к этой программе. В ней должно быть по одной двойственной переменной на каждое ребро сети и на каждую вершину, кроме S и T.

Теперь мы решим ту же задачу в общем виде. Вспомним линейную программу для общей задачи максимального потока (раздел 7.2).

(c) Запишите программу, двойственную к ней, используя переменные y_e для каждого ребра e, а также x_u для каждой вершины $u \neq s, t$.

(d) Покажите, что всякое решение этой двойственной программы обладает таким свойством: сумма значений y_e вдоль любого ориентированного пути из s в t не меньше 1.

(e) Каков интуитивный смысл двойственных переменных? Покажите, что всякий (s,t)-разрез в сети можно преобразовать в некоторое допустимое

двойственное решение, стоимость которого в точности равна пропускной способности этого разреза.

7.26. В совместной системе линейных неравенств

$$a_{11}x_1 + \ldots + a_{1n}x_n \leqslant b_1,$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{m1}x_1 + \ldots + a_{mn}x_n \leqslant b_m$$

мы называем j-е неравенство *вынужденным равенством* (forced-equal), если оно обращается в равенство для *всякого* допустимого решения $\mathbf{x} = (x_1, \ldots, x_n)$. Другими словами, $\sum_i a_{ji}x_i \leqslant b_j$ *не является* вынужденным равенством, если существует некоторое $\mathbf{x}$, которое удовлетворяет всем ограничениям, и при этом $\sum_i a_{ji}x_i < b_j$.

Например, в случае

$$x_1 + x_2 \leqslant 2,$$
$$-x_1 - x_2 \leqslant -2,$$
$$x_1 \leqslant 1,$$
$$-x_2 \leqslant 0$$

первые два неравенства являются вынужденными равенствами, а третье и четвёртое — нет.

Решение $\mathbf{x}$ системы неравенств называется *характеристическим* (characteristic), если для всякого неравенства I, не являющегося вынужденным равенством, $\mathbf{x}$ удовлетворяет I, не обращаясь в равенство. В примере выше таким решением является $(x_1, x_2) = (-1, 3)$, для которого $x_1 < 1$ и $-x_2 < 0$, в то время как $x_1 + x_2 = 2$ и $-x_1 - x_2 = -2$.

(a) Покажите, что всякая совместная система имеет характеристическое решение.

(b) Покажите, как при помощи линейного программирования можно узнать, какие неравенства в данной совместной системе являются вынужденными равенствами, и найти характеристическое решение.

7.27. Покажите, что *задача о размене монет* (упражнение 6.17) может быть сформулирована как целочисленная линейная программа. Можем ли мы решать соответствующую задачу линейного программирования, будучи уверенными, что решение окажется целочисленным (как в случае двудольного паросочетания)? Докажите это или приведите контрпример.

7.28. *Линейная программа для кратчайшего пути.* Пусть мы хотим найти кратчайший путь из вершины s в вершину t в ориентированном графе с длинами рёбер $l_e > 0$.

(a) Покажите, что это эквивалентно поиску (s, t)-потока f, который минимизирует $\sum_e l_e f_e$ с учётом ограничения $\mathrm{size}(f) = 1$. Ограничений на пропускную способность нет.

(b) Запишите задачу о кратчайшем пути в виде линейной программы.

(c) Покажите, что двойственную линейную программу можно записать в виде

$$x_s - x_t \to \max$$
$$x_u - x_v \leqslant l_{uv} \text{ для всех } (u, v) \in E.$$

(d) Мы уже обсуждали двойственную программу для задачи о кратчайшем пути на с. 205. В чём разница между теперешней двойственной программой и тогдашней?

7.29. *Голливуд.* Кинопродюсер ищет актёров и инвесторов для своего нового фильма. Есть n актёров; i-й актёр согласен сниматься за s_i долларов. Есть также m инвесторов; j-й инвестор предоставляет p_j долларов, но с условием, что *все* актёры из определённого множества $L_j \subseteq \{1, 2, \ldots, n\}$ будут сниматься в фильме.

Выручка продюсера — это сумма выплат инвесторов минус выплаты актёрам. Цель всякого продюсера — максимизировать выручку.

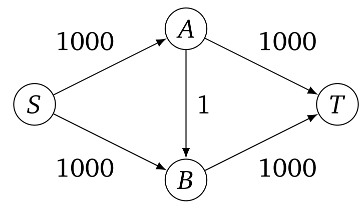
(a) Сформулируйте это в виде задачи целочисленного программирования, в которой переменные принимают значения из множества $\{0, 1\}$. (Введите переменную x_i для участия i-го актёра и переменную y_j для участия j-го спонсора. Как записать условия с помощью неравенств вида $y_j \leqslant x_i$?)

(b) Покажите, что возникающая линейная программа в действительности имеет *целочисленное* оптимальное решение (как и в случае с максимальным потоком и паросочетанием в двудольном графе), и что нецелое оптимальное решение (найденное с помощью линейного программирования) легко преобразовать в целое. (Подсказка: те переменные, которые не равны нулю или единице, можно все вместе сдвигать, пока одна из них не упрётся в край, так мы уменьшаем число нецелых значений, сохраняя оптимальность.)

7.30. *Теорема Холла.* Возвратимся к танцам из раздела 7.3. Предположим, что у нас имеется двудольный граф с каким-то количеством мальчиков слева и таким же количеством девочек справа. Теорема Холла утверждает, что совершенное паросочетание существует тогда и только тогда, когда для любых k мальчиков есть по крайней мере k девочек, соединённых с кем-то из выбранных k мальчиков.

Докажите эту теорему. (Подсказка: можно воспользоваться теоремой Форда—Фалкерсона.)

7.31. Рассмотрите следующую простую сеть с указанными пропускными способностями рёбер.



(a) Покажите, что если запустить для этого графа алгоритм Форда—Фалкерсона, то неразумный выбор путей в остаточной сети может привести к тысяче итераций! (Представьте себе, что было бы, если бы пропускные способности были равны миллиону.)

Сейчас мы укажем способ выбирать пути, при котором алгоритм гарантированно останавливается за разумное число итераций.

Рассмотрим произвольную ориентированную сеть $G = (V, E)$, s, t, $\{c_e\}$, в которой мы хотим найти максимальный поток. Предположим, что все пропускные способности рёбер — целые числа, не меньшие 1, и назовём пропускной способностью пути из s в t наименьшую пропускную способность рёбер этого пути. Самым *широким* (fattest) путём из s в t назовём путь с наибольшей пропускной способностью.

(b) Покажите, что самый широкий путь из s в t можно найти с помощью (модифицированного) алгоритма Дейкстры.

(c) Покажите, что максимальный поток в G есть сумма отдельных потоков вдоль не более чем $|E|$ путей из s в t.

(d) Теперь покажите, что если мы всегда увеличиваем поток вдоль самого широкого пути в остаточной сети, то алгоритм Форда—Фалкерсона нахождения максимального потока остановится не более чем за $O(|E| \log F)$ итераций, где F — размер максимального потока. (Подсказка: полезно вспомнить доказательство теоремы о жадном покрытии множествами в разделе 5.4.)

На самом деле, ещё более простой подход — построение пути в остаточном графе при помощи поиска в ширину — гарантирует, что потребуется не более $O(|V| \cdot |E|)$ итераций.

Глава 8

NP-полные задачи

8.1. Задачи поиска

В семи предыдущих главах мы рассмотрели алгоритмы поиска кратчайших путей и минимальных покрывающих деревьев, паросочетаний в двудольных графах, максимальных возрастающих последовательностей, максимальных потоков в сетях и так далее. Все эти алгоритмы *эффективны* в том смысле, что с ростом длины входа (n) время работы растёт не быстрее, чем полином (скажем, n, n^2 или n^3) от n.

Чтобы понять, в чём тут достижение, заметим, что во всех этих задачах решение (путь, дерево, последовательность) ищется среди *экспоненциального* числа кандидатов. Действительно, есть $n!$ способов разбить n девочек и n мальчиков на пары, в полном графе с n вершинами есть n^{n-2} покрывающих деревьев, в графе обычно есть экспоненциальное число путей из s в t. Поэтому если просто пробовать все варианты, то получится алгоритм с экспоненциальным временем работы, скорее всего бесполезный на практике (см. следующую врезку). Эффективному алгоритму, таким образом, необходимо каким-то хитрым способом избежать такого полного перебора, используя свойства решаемой задачи.

Мы уже рассмотрели приёмы, позволяющие в некоторых случаях избежать экспоненциального перебора: жадные алгоритмы, динамическое программирование, линейное программирование (а вот метод «разделяй и властвуй», как правило, лишь ускоряет решение задач, которые и без него

От Сиссы до Мура

Легенда говорит, что брамин Сисса, придумав шахматы, показал их начальству (королю) — которое игру одобрило и предложило выбрать награду. Брамин скромно попросил положить одну рисинку на первую клетку доски, две — на вторую, четыре — на третью и так далее, удваивая количество вплоть до шестьдесят четвёртой клетки. Король неосторожно согласился и вошёл в историю как первая жертва экспоненциального роста («комбинаторного взрыва»): запрошенное количество риса, $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ зёрен, в тысячи раз превосходит годовой урожай риса во всей Индии! Это не просто байки: скажем, бактерии (при отсутствии ограничений) тоже размножаются экспоненциально. В 1798 году английский философ Томас Роберт Мальтус предсказал, в со-

временных терминах, что экспоненциально растущее (он называл это «геометрическим ростом») население неизбежно сталкивается с полиномиально ограниченными ресурсами (идея, позже повлиявшая на Дарвина).

В 1965 году Гордон Мур, один из первых разработчиков микросхем, заметил, что количество транзисторов в микросхеме в 60-х годах удваивалось примерно каждый год. Мур предположил, что так и будет продолжаться. Сейчас «закон Мура» обычно формулируют так: быстродействие компьютера удваивается каждые 18 месяцев. И действительно, это происходит уже лет сорок — и наряду с разработкой эффективных алгоритмов стало основой стремительного развития информационных технологий.

На первый взгляд кажется, что закон Мура делает ненужными полиномиальные алгоритмы: пусть даже время работы алгоритма экспоненциально велико, почему бы просто не подождать, пока компьютеры станут достаточно быстрыми? На самом деле наоборот: рост быстродействия важен именно потому, что мы используем полиномиальные алгоритмы.

Для примера рассмотрим алгоритм, решающий задачу выполнимости за время $O(2^n)$. В 1975 году такой алгоритм за час мог бы решить задачу для 25 переменных, в 1983 — для 31, в 1995 — для 38. Наконец, на современных компьютерах этот алгоритм за час смог бы обрабатывать формулы с 45 переменными. Прогресс налицо, да только появление новой переменной (удвоение числа операций) требует полутора лет ожидания, а размеры формул в приложениях (в том числе используемых при разработке процессоров) растут гораздо быстрее.

А вот для алгоритма с асимптотикой $O(n)$ или $O(n \log n)$ допустимый размер входов *увеличивается примерно в 100 раз* каждые 10 лет, для алгоритма со временем работы $O(n^2)$ — в 10 раз. Даже не особо эффективный алгоритм с оценкой $O(n^6)$ позволит за это время удвоить размер решаемых задач. Можно сказать так: экспоненциальный рост скорости процессоров приводит к полиномиальному росту размера задач в случае экспоненциальных алгоритмов, и к экспоненциальному росту размера задач для полиномиальных алгоритмов. Так что воспользоваться законом Мура мы смогли именно благодаря эффективным алгоритмам.

Как понимали и Сисса, и Мальтус, и Мур, экспоненциальному росту рано или поздно приходит конец — бактерии останутся без еды, а микросхемы упрутся в атомный масштаб, закону Мура осталось десять или двадцать лет. Дальше будут нужны новые алгоритмы или же совсем новые идеи — скажем, *квантовые вычисления*, о которых пойдёт речь в главе 10.

решаются за полиномиальное время). Сейчас же настало время познакомиться с задачами, для которых эффективного решения до сих пор не известно. В них тоже надо найти объект с некоторыми свойствами среди экспоненциально большого множества кандидатов — но для них никто не знает, как избежать перебора: самые быстрые известные алгоритмы для таких задач имеют экспоненциальное время работы.

Задача выполнимости

Задача выполнимости (Satisfiability, SAT) (см. упражнение 3.28 и раздел 5.3) важна и практически (тестирование микросхем, анализ изображений и др.), и теоретически — как эталонный пример трудной задачи.

На вход подаётся формула вроде такой:

$$(x \vee y \vee z) \quad (x \vee \bar{y}) \quad (y \vee \bar{z}) \quad (z \vee \bar{x}) \quad (\bar{x} \vee \bar{y} \vee \bar{z}).$$

Это *булева формула в конъюнктивной нормальной форме* (Boolean formula in conjunctive normal form, CNF). Она представляет собой конъюнкцию[1] (логическое «и») нескольких *дизъюнктов* (clauses), каждый из которых является дизъюнкцией (логическое *или*, обозначаемое через $\vee$) *литералов*. Литералом называется булева переменная (например, x) или её отрицание ($\bar{x}$). *Выполняющим набором* (satisfying truth assignment) называется набор значений переменных («истина» и «ложь», `true` и `false`), при которых формула истинна, то есть каждый дизъюнкт содержит хотя бы один истинный литерал. Задача выполнимости заключается в том, чтобы по данной булевой формуле в КНФ либо найти её выполняющий набор, либо сообщить, что таких наборов нет.

Например, если присвоить значение `true` всем переменным формулы, рассмотренной выше, то будут выполнены все дизъюнкты, кроме последнего. Есть ли набор, который выполняет *все* дизъюнкты?

Немного подумав, можно понять, что у этой формулы выполняющего набора нет. (Подсказка: если три средних дизъюнкта выполнены, то значения всех трёх переменных равны.) Но как искать выполняющий набор в общем случае? Можно, конечно, просто перебрать все возможные наборы, но для формул с n переменными их будет 2^n, то есть экспоненциально много.

Задача выполнимости — типичная *задача поиска* (search problem). Нам дан *вход* (instance) I, то есть входные данные, задающие условие задачи (в нашем случае это булева формула в КНФ), и требуется найти *решение* (solution) S — объект, удовлетворяющий некоторым ограничениям; в нашем случае выполняющий набор. Если решения не существует, мы должны сообщить об этом.

Важно, что каждого кандидата (каждый набор значений) можно проверить быстро: его запись в виде строки битов имеет полиномиальный размер, а проверка годности кандидата требует полиномиального числа операций. (В обоих случаях мы имеем в виду полином от длины входа.) В нашей задаче число переменных не больше длины входа, а проверка состоит в вычислении значения формулы, что тоже несложно.

В дальнейшем нам будет удобно считать, что задача поиска *задаётся* таким алгоритмом проверки.

> *Задача поиска* (search problem) задаётся алгоритмом $\mathscr{C}$ с двумя аргументами I (условие задачи) и S (кандидат), время работы которого ограничено полиномом от $|I|$. Говорим, что S является *решением* для входа I, если $\mathscr{C}(I, S) = \texttt{true}$.

[1] Обычно конъюнкцию обозначают знаком $\wedge$, но мы его опускаем, как делается в алгебре для знака умножения.

Несмотря на усилия множества исследователей в течение последних пятидесяти лет, все известные алгоритмы для задачи выполнимости имеют экспоненциальное (в худшем случае) время работы. С другой стороны, для двух её естественных частных случаев известны эффективные (полиномиальные) алгоритмы.

(1) Если каждый дизъюнкт содержит не более одного положительного литерала, то формула называется *хорновской* и её выполнимость может быть проверена простым жадным алгоритмом из раздела 5.3.

(2) Если все дизъюнкты содержат не более *двух* литералов, то помогает теория графов: задачу можно решить при помощи поиска компонент сильной связности в специальном графе, построенном по входной формуле (упражнение 3.28). Ещё один алгоритм для этого частного случая см. в главе 9.

Но чуть ослабив ограничения и разрешив каждому дизъюнкту содержать до *трёх* литералов (как в примере выше), мы опять получаем трудную задачу!

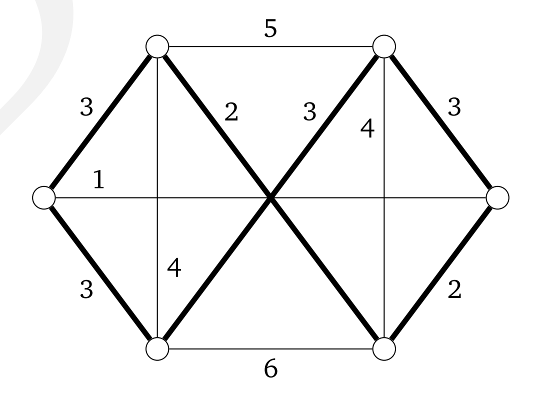
Задача коммивояжёра

В *задаче коммивояжёра* (traveling salesman problem, TSP) даны n вершин $1, \ldots$ $\ldots, n$ и все $n(n-1)/2$ расстояний между ними, а также некоторый бюджет b. Найти необходимо *маршрут*, то есть цикл, который проходит по каждой вершине графа ровно один раз, общей стоимостью не более b (или же сообщить, что такого цикла нет). Другими словами, мы ищем перестановку $\tau(1), \ldots, \tau(n)$ вершин, для которой длина обхода в этом порядке не превосходит b:

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \ldots + d_{\tau(n),\tau(1)} \leqslant b.$$

Пример приведён на рис. 8.1 (нарисованы не все рёбра; считаем, что остальные длины очень велики).

Рис. 8.1. Оптимальный маршрут коммивояжёра, выделенный жирным, имеет длину 16.



Отдельно отметим, что мы сформулировали задачу коммивояжёра как *задачу поиска*: по данному условию найти маршрут в пределах данного бюджета или сообщить, что такого нет. Ясно, что на самом деле это *оптимизационная задача*, в которой нас интересует самый *дешёвый* маршрут. Зачем же мы рассмотрели другую, менее естественную формулировку?

На то есть причина: мы хотим сравнивать и связывать друг с другом различные задачи, и чтобы сравнивать оптимизационные задачи с задачами поиска (например, с выполнимостью), нам удобно формулировать оптимизационные задачи как задачи поиска. Превращение оптимизационной задачи в задачу поиска не меняет её сложности, потому что обе эти задачи *сводятся друг к другу*. Умея искать самый дешёвый маршрут, мы можем проверить, укладывается ли он в бюджет. Наоборот, алгоритм для задачи поиска может быть использован для нахождения оптимального маршрута: для этого не хватает информации о его стоимости, но её можно найти с помощью бинарного поиска (упражнение 8.1).

Здесь, однако, есть тонкость: а зачем мы ввели бюджет, почему бы не сказать по-простому, что мы ищем оптимальный маршрут? Дело в том, что если считать решением задачи оптимальный маршрут, то проверка кандидата перестанет быть полиномиальной: как узнать, оптимален ли предложенный маршрут, не перебирая все остальные? А проверить, что маршрут укладывается в бюджет, легко.

Как и для выполнимости, для задачи коммивояжёра (несмотря на многолетние попытки) так и не удалось придумать полиномиальный по времени алгоритм. В разделе 6.6 мы видели, что не обязательно перебирать все $(n-1)!$ маршрутов, если воспользоваться динамическим программированием, но алгоритм остаётся экспоненциальным.

Поучительно сравнить задачу коммивояжёра с задачей о минимальном покрывающем дереве, для которой *есть* эффективные алгоритмы. Эту задачу можно переформулировать в виде задачи поиска: по данной матрице расстояний и бюджету b найти дерево T общего веса $\sum_{(i,j)\in T} d_{ij} \leqslant b$. Если запретить дереву ветвиться, то получится задача коммивояжёра.[1] Эта маленькая деталь мгновенно делает задачу гораздо более сложной.

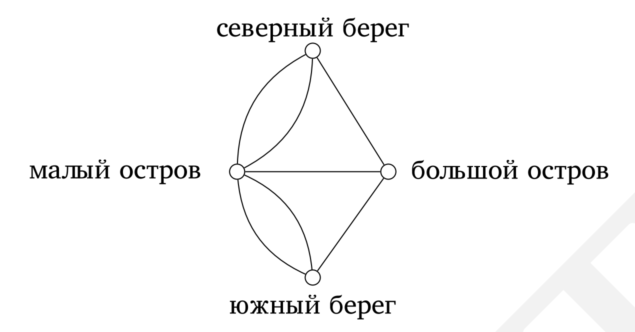
Эйлер и Рудрата

Летом 1735 года Леонард Эйлер, известный швейцарский (и российский: он работал в Петербурге) математик, прогуливался по мостам города Кёнигсберга (ныне Калининград). Через некоторое время он обнаружил, что независимо от того, где он начинает свою прогулку и как именно он идёт, не получается пройти по каждому мосту ровно один раз — и так появилась первая работа по теории графов.

Что же мешает построению такого маршрута? Преобразуем карту в граф, вершинами которого будут участки суши (два острова и два берега), а рёб-

[1] Точнее, получится задача о поиске пути, а не цикла, но можно показать, что их сложности почти не отличаются.

рами — соединяющие их мосты:



В графе есть кратные рёбра, отвечающие за разные мосты между двумя вершинами. Мы хотим найти путь, который проходит по всем *рёбрам* ровно по одному разу (путь может проходить через некоторые вершины больше одного раза). Другими словами, нас интересует, *можно ли нарисовать граф, не отрывая пера от бумаги*.

Эйлер придумал простой и изящный ответ на этот вопрос: *необходимо и достаточно, чтобы граф был связен и чтобы степень всех вершин, кроме, возможно, двух* (*начала и конца пути*)*, была чётна* (упражнение 3.26). Это объясняет, почему нельзя обойти все мосты Кёнигсберга по одному разу: степени всех четырёх вершин нечётны.

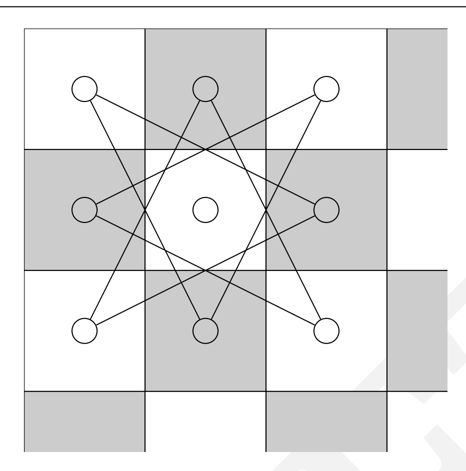
В терминах задач поиска: входом *задачи об эйлеровом пути* (Euler path) является граф, а решением — путь, проходящий по каждому ребру ровно по одному разу. Из критерия Эйлера следует, что эта задача может быть решена за полиномиальное время (рассуждение Эйлера указывает искомый путь).

Почти за тысячу лет до этого кашмирский поэт по имени Рудрата задал следующий вопрос: можно ли ходом коня обойти все клетки шахматной доски, побывав в каждой клетке ровно один раз, и вернуться в исходную клетку? Снова получаем задачу на графе: в нём 64 вершины (клетки); две клетки соединены ребром, если за один ход конём можно из одной попасть в другую (в таком случае координаты этих двух клеток по одному направлению различаются на 2, а по другому — на 1). На рис. 8.2 показана часть графа, соответствующая левому верхнему углу доски. Можете ли вы найти искомый маршрут?

Как видите, здесь требуется обойти по разу все *вершины* графа, а не все рёбра как в задаче Эйлера. Аналогичный вопрос может быть поставлен не только для шахматной доски, но и для любого графа. Получаем *задачу о гамильтоновом цикле*: найти в данном графе цикл, проходящий через все вершины графа по одному разу, или сообщить, что такого нет.[1] Задача очень похожа на задачу коммивояжёра, и полиномиальных алгоритмов для неё неизвестно.

[1] В оригинале книги авторы называют данную задачу задачей Рудраты и замечают, что её в XIX веке переоткрыл ирландский математик Гамильтон — тот самый, в честь которого названы гамильтонианы в физике. В русском переводе сохранено традиционное название — *задача о гамильтоновом цикле*. — *Прим. перев.*

Рис. 8.2. Ходы коня в углу доски.



В задаче о эйлеровом пути необходимо обойти все *рёбра*, в то время как в задаче о гамильтоновом цикле нужно обойти все *вершины*. Но есть ещё и другое различие: в первой мы ищем путь, а во второй — цикл. Это, однако, не принципиально: можно искать *гамильтонов путь*, проходящий все вершины по одному разу, и эта задача так же сложна, как и задача о цикле (см. ниже).

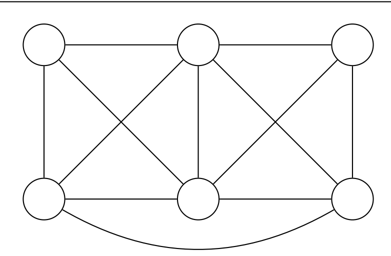
Разрезы и бисекции

Разрезом (cut) называется множество рёбер, при выкидывании которого граф теряет связность. В *задаче о минимальном разрезе* (minimum cut) по данному графу и бюджету b необходимо найти разрез из b (или меньше) рёбер. Например, минимальный разрез графа рис. 8.3 имеет вес 3. Эту задачу можно решить, вычислив максимальный поток $n-1$ раз: присвоим каждому ребру графа пропускную способность 1, зафиксируем какую-нибудь вершину графа и найдём максимальный поток из этой вершины до каждой из оставшихся вершин. Минимальная величина этих потоков будет равна (по теореме о максимальном потоке и минимальном разрезе) минимальному разрезу. (Мы видели совсем другой вероятностный алгоритм для этой задачи на с. 137.)

Для многих графов (в том числе графа рис. 8.3) минимальный разрез отсоединяет от графа одну вершину (состоит из всех рёбер, инцидентных какой-то вершине графа). Более интересны разрезы, которые разделяют вершины графа на две сопоставимые по размеру части. В *задаче о сбалансированном разрезе* (balanced cut) по данному графу с n вершинами и бюджету b необходимо найти разбиение вершин графа на два множества S и T, для которого $|S|, |T| \geqslant n/3$ и множества S и T соединены не более чем b рёбрами. Очередная трудная задача…

Вот пример практической ситуации, где полезны сбалансированные разрезы. Пусть на картинке нужно выделить объекты (скажем, отделить слона от

Рис. 8.3. Каков минимальный разрез в этом графе?



зелёной травы и синего неба). Построим граф, где каждому пикселю изображения будет соответствовать вершина и где два пикселя соединены ребром, если они расположены рядом и имеют похожие цвета. Тогда отдельному объекту картинки (слону, например) будет соответствовать множество сильно связанных друг с другом (и слабо связанных с остальной частью) вершин. Минимальный сбалансированный разрез разделит вершины на два кластера, не разрезав объекты — скажем, сначала он может отделить слона от неба и травы, а последующий разрез второй части разделит небо и траву. Подобные задачи называют *кластеризацией*.

Целочисленное линейное программирование

Как мы уже видели в главе 7, симплекс-метод не полиномиален по времени, но известны и полиномиальные алгоритмы решения задач линейного программирования, так что эта задача эффективно решается и в теории, и на практике. Хуже обстоит дело, если нас интересуют только целые значения переменных. Эта задача называется *задачей целочисленного линейного программирования* (ЦЛП; integer linear programming, ILP). В виде задачи поиска она выглядит так: дано множество линейных неравенств $\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$, где $\mathbf{A}$ — матрица размера $m \times n$, а $\mathbf{b}$ — вектор длины m; кроме того, дана целевая функция, заданная вектором $\mathbf{c}$ размера n, а также *цель* g (аналог бюджета в максимизационных задачах). Необходимо найти неотрицательный *целочисленный* вектор $\mathbf{x}$ длины n, для которого $\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$ и $\mathbf{c} \cdot \mathbf{x} \geqslant g$.

Как видно, нужно найти вектор, удовлетворяющий некоторым линейным неравенствам (в нашей формулировке ограничение $\mathbf{c} \cdot \mathbf{x} \geqslant g$ можно присоединить к системе неравенств $\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$). Таким образом, по данным $\mathbf{A}$ и $\mathbf{b}$ нужно найти вектор $\mathbf{x}$, для которого $\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$, или же сообщить, что такого нет. Эффективный алгоритм для такой задачи был бы чрезвычайно полезен на практике, но такой алгоритм до сих пор не придумали.

Есть также очень специальный случай этой задачи, который уже труден: необходимо найти вектор $\mathbf{x}$, состоящий только из нулей и единиц, для которого $\mathbf{A}\mathbf{x} = \mathbf{1}$, где $\mathbf{A}$ — матрица размера $m \times n$, состоящая из нулей и единиц, а $\mathbf{1}$ — вектор длины m из одних единиц. Это действительно частный случай (см. раздел 7.1.4); назовём его *задачей об уравнениях в нулях и единицах* (УНЕ; zero-one equations, ZOE).

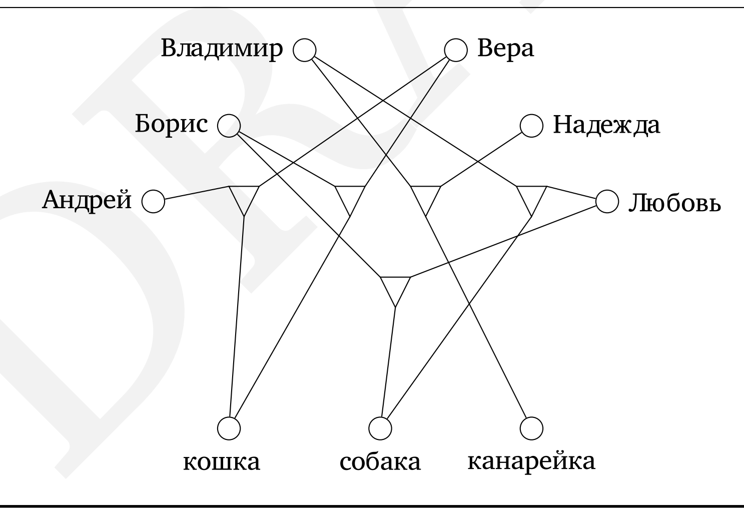
Мы привели несколько примеров задач поиска: одни решаются за полиномиальное время, а другие, хотя и похожие, до сих пор такому решению не поддаются. Мы приведём ещё несколько задач, которые нам понадобятся дальше, при сравнении сложности задач; нетерпеливые читатели могут пока это пропустить и перейти к разделу 8.2 (и возвращаться, когда эти задачи будут упоминаться).

Трёхдольное сочетание

Вспомним задачу о паросочетании в двудольном графе: дан двудольный граф с долями размера n (условно говоря, n мальчиков и n девочек); требуется найти n рёбер, не имеющих общих вершин, или сообщить, что таких нет. В разделе 7.3 мы видели, как решить эту задачу, сведя её к задаче о максимальном потоке. Но для более общей *задачи трёхдольного сочетания* (3D Matching) полиномиальный алгоритм придумать не удаётся. В новой постановке кроме n мальчиков и n девочек есть ещё и n домашних животных, а вместо рёбер задаются *тройки*, состоящие из мальчика, девочки и животного. Тройка (b, g, p) означает, что мальчик b, девочка g и животное p уживутся друг с другом. Мы хотим найти n непересекающихся троек, создав тем самым n счастливых «семей».

Видите ли вы решение на рис. 8.4?

Рис. 8.4. Пример задачи о трёхдольном сочетании. Треугольники обозначают допустимые варианты.

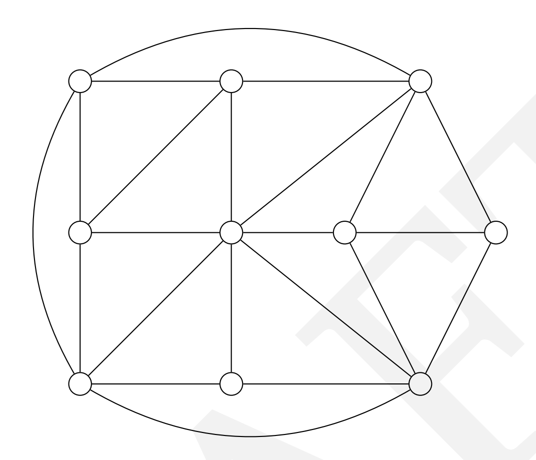


Независимое множество, вершинное покрытие и клика

В *задаче о независимом множестве* (independent set problem), которую мы уже рассматривали в разделе 6.7, на вход даётся граф и число g, а найти необходимо g вершин, никакие две из которых не соединены ребром. Можете ли

вы найти три такие вершины в графе на рис. 8.5? А четыре? Из раздела 6.7 мы знаем, что для дерева эта задача простая — но для произвольных графов полиномиальный алгоритм неизвестен.

Рис. 8.5. Каково максимальное независимое множество в этом графе?



Есть много других задач поиска для графов. В *задаче о вершинном покрытии* (vertex cover problem) для данного графа и числа b требуется найти b вершин, которые «задевают» каждое ребро (для любого ребра графа хотя бы один из концов ребра лежит в этом множестве). Можете ли вы покрыть все рёбра графа рис. 8.5 семью вершинами? А шестью? Как связана эта задача с задачей о независимом множестве?

Задача о вершинном покрытии (vertex cover problem) является частным случаем задачи о покрытии множествами (см. главу 5). В задаче о покрытии множествами на вход даются множество E и несколько его подмножеств $S_1, \ldots, S_m$, а также число b. Найти необходимо b из заданных подмножеств, объединение которых покрывает E. (Как свести задачу о трёхдольном сочетании к задаче о покрытии множествами?)

В задаче о вершинном покрытии элементами E являются рёбра графа, а для каждой вершины графа есть множество S_i, содержащее все смежные с ней рёбра.

Наконец, в *задаче о клике* (clique problem) по данному графу и числу g необходимо найти множество из g вершин, каждые две из которых соединены ребром. Каков размер максимальной клики в графе на рис. 8.5? Как задача о клике связана с задачей о независимом множестве?

Максимальный путь

Мы уже знаем, что задача о кратчайшем пути в графе решается эффективно. А что можно сказать про *задачу о максимальном пути* (longest path problem)?

В ней дан граф G с неотрицательными весами на рёбрах и двумя выделенными вершинами s и t, а также число g. Найти необходимо путь из s в t суммарного веса не меньше g. Естественно, мы предполагаем, что путь должен быть простым (не заходить дважды в одну и ту же вершину).

Эффективных алгоритмов для этой задачи (которой могли бы заинтересоваться жулики-таксисты) тоже пока не нашли.

Рюкзак и сумма подмножества

Вспомним *задачу о рюкзаке* (knapsack problem), уже рассмотренную нами в разделе 6.4. В ней заданы целочисленные веса $w_1, \ldots, w_n$ и стоимости $v_1, \ldots, v_n$ для n предметов. Также задана граница W для веса и цель g (первый параметр есть в исходной оптимизационной задаче, второй параметр добавлен для того, чтобы получить задачу поиска). Необходимо выбрать несколько предметов так, чтобы их суммарный вес не превышал W и чтобы в то же время их суммарная стоимость была не меньше g. Как обычно, если такой возможности нет, требуется сообщить об этом.

С помощью динамического программирования в разделе 6.4 мы разработали алгоритм для задачи о рюкзаке со временем работы $O(nW)$. Мы отмечали, что это экспоненциальное время работы, потому что в качестве множителя оно включает W, а не $\log W$ (размер двоичной записи числа W). Есть и другой алгоритм с экспоненциальным временем работы: просто переберём все 2^n возможных вариантов выбора. А вот существует ли для этой задачи полиномиальный алгоритм, неизвестно.

Рассмотрим теперь вариант задачи о рюкзаке, в котором числа на вход даются в *унарной* системе счисления: например, число 12 запишется как $IIIIIIIIIIII$. Ясно, что неразумно записывать числа таким образом, но тем не менее мы имеем полное право определить такую задачу поиска, которую будем называть *унарной задачей о рюкзаке* (unary knapsack problem). Мы уже знаем, что эта (искусственная) задача решается за полиномиальное время.

А вот ещё один вариант: пусть вес каждого предмета равен его стоимости (и все числа даны в двоичной системе счисления), а также $g = W$. Продолжая историю с вором, здесь можно сказать, что все предметы представляют собой золотые слитки (разного веса) и вор хочет забить ими свой рюкзак под завязку. Ясно, что в этой задаче нам просто надо найти в данном множестве (или даже мультимножестве, если есть повторяющиеся числа) подмножество с суммой W. Это частный случай задачи о рюкзаке — быть может, хотя бы для него есть полиномиальный алгоритм? Оказывается, что эта задача, называемая *задачей о сумме подмножества* (subset sum problem), тоже очень трудна.

Но раз уж задача о сумме подмножества не проще общей задачи о рюкзаке, почему этот специальный случай интересен? Потому что он *просто формулируется*, и когда мы будем сводить задачи поиска друг к другу, такие простые эталонные задачи, как задача о сумме подмножества и задача 3-выполнимости, будут очень полезны.

8.2. NP-полные задачи: определения и примеры

Простые и трудные задачи

Как мы видели, небольшое изменение условия может радикально упростить задачу. Вот некоторые примеры:

трудные задачи (NP-полные)	простые задачи (из P)
3-выполнимость	2- (или хорновская) выполнимость
коммивояжёр	минимальное покрывающее дерево
максимальный путь	кратчайший путь
трёхдольное сочетание	двудольное паросочетание
рюкзак	унарный рюкзак
независимое множество	независимое множество в деревьях
ЦЛП	ЛП
гамильтонов путь	эйлеров путь
сбалансированный разрез	минимальный разрез

В правом столбце приведены задачи, для которых мы уже знаем эффективные алгоритмы. А вот для задач левого столбца их не удаётся придумать в течение нескольких десятилетий.

Разные простые задачи просты по-своему (где-то помогает динамическое программирование, где-то потоки в сетях, поиск в графе, жадные алгоритмы). А вот все задачи левого столбца сложны по одной и той и причине: все эти задачи *эквивалентны*, любую из них можно свести к любой другой. Можно сказать, что это по существу одна задача в разных вариантах.

Классы P и NP

Мы готовы определить два важных класса задач. Напомним, что задача поиска задаётся алгоритмом $\mathscr{C}$, который по входу задачи I и кандидату S выдаёт `true` тогда и только тогда, когда S является решением для I. Время работы $\mathscr{C}(I, S)$ должно быть ограничено полиномом от длины входа $|I|$. *Через NP мы обозначаем класс всех задач поиска.*

Мы уже знакомы со многими задачами поиска, которые решаются за полиномиальное время. Для таких задач существует алгоритм, который получает на вход условие I и за полиномиальное от $|I|$ время либо выдаёт решение, либо сообщает, что решения нет. *Класс всех задач поиска, которые могут быть решены за полиномиальное время, обозначается через P.*

Существуют ли задачи поиска, которые не могут быть решены за полиномиальное время? Другими словами, верно ли, что $\mathrm{P} \neq \mathrm{NP}$? Большинство специалистов полагают, что да — трудно представить себе, что всегда можно избежать экспоненциального перебора и что существует (до сих пор не открытый, несмотря на многолетние усилия) метод, позволяющий решать все задачи из NP. Другая причина: поиск не слишком длинных доказательств матема-

Почему P и NP?

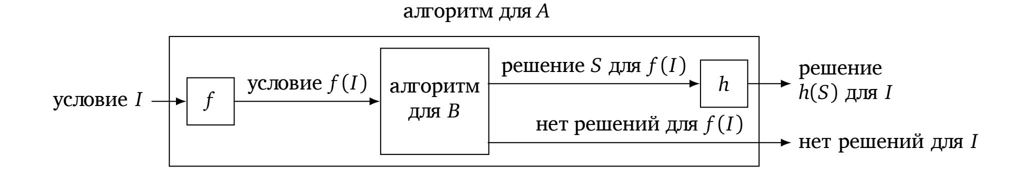
Легко догадаться, что P происходит от слова «полиномиальный» (polynomial). Но откуда взялось NP? Не от «no problem» (совсем наоборот!), а от слов «nondeterministic polynomial time» — недетерминированное полиномиальное время. В первоначальном определении класса NP рассматривались «недетерминированные алгоритмы», которые каким-то чудом могли угадывать, какое действие им нужно совершить. Класс NP исходно определялся не как класс задач поиска, а как класс *задач разрешения* (decision problems), требующих ответа да/нет. (Пример: существует ли набор, выполняющий данную формулу? Недетерминированный алгоритм угадывал и проверял такой набор за полиномиальное время.) Тогда такие задачи были более привычными в силу традиции теории формальных языков.

тических теорем лежит в NP (проверка записанного формально доказательства полиномиальна), так что равенство P и NP в каком-то смысле означало бы, что математики не нужны. Есть и другие причины думать, что $P \neq NP$— но вопрос о равенстве классов P и NP остаётся одной из важнейших нерешённых математических проблем.

Ещё раз о сведéниях

Пусть мы согласились, что $P \neq NP$ — но почему сложны конкретные задачи в левом столбце нашей таблицы? Дело в том, что в некотором смысле все эти задачи эквивалентны (сводятся одна к другой); более того, они максимально сложны в классе NP: если бы какая-то из них решалась за полиномиальное время, то и вообще все NP-задачи решались бы за полиномиальное время. Чтобы доказать это, используют *сведéния* (reductions), которые переписывают одну задачу в терминах другой.

В главе 7 мы уже видели примеры сведения одной задачи к другой. Приведём точное определение для задач поиска. *Сведением* задачи поиска A к задаче поиска B называется пара полиномиальных алгоритмов f и h с такими свойствами: чтобы найти решение задачи A для входа I, достаточно решить задачу B для входа $f(I)$ и к полученному решению S применить алгоритм h (см. диаграмму ниже). А если у B для входа $f(I)$ нет решения, то не должно быть решения и у A для входа I. Имея такие алгоритмы f и h, мы можем приспособить алгоритм задачи B для решения задачи A (сохраняя полиномиальность).



Два взгляда на сводимость

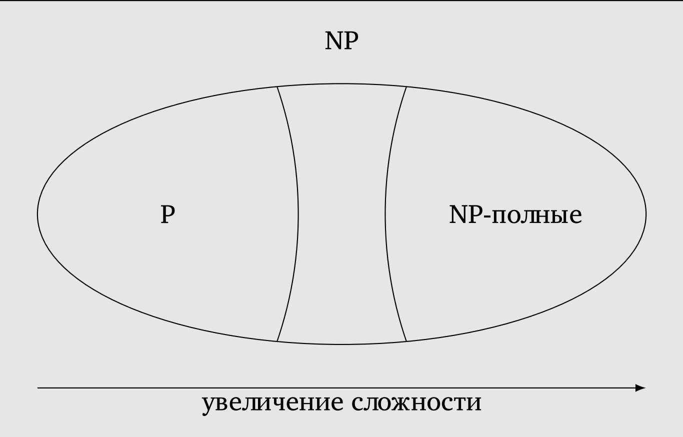
С точки зрения практика сведение задачи A к B позволяет быстро решать задачу A, используя готовые быстрые алгоритмы для B. Но в этой главе мы подходим к сведениям как теоретики и замечаем, что если трудная задача A сводится к B, то и B трудна.

Обозначая сведение A к B через

$$A \to B,$$

будем говорить, что *сложность* увеличивается в направлении стрелки, а эффективные алгоритмы переносятся с одной задачи на другую в обратном направлении (рисунок 8.6). Из любой NP-задачи ведёт стрелка к NP-полной задаче, поэтому полиномиальный алгоритм для (какой-либо) NP-полной задачи распространился бы на все NP-задачи.

Рис. 8.6. Класс NP содержит класс P. Мы не умеем доказывать, что они не совпадают. Но про некоторые задачи мы знаем, что они NP-полны (самые трудные в NP): если P $\neq$ NP, то эти (NP-полные) задачи заведомо не лежат в P. Известно также, что если P $\neq$ NP, то найдутся и промежуточные задачи (не в P и не NP-полные).



Сведения *транзитивны*:

$$\text{если} \quad A \to B \quad \text{и} \quad B \to C, \quad \text{то} \quad A \to C.$$

Чтобы убедиться в этом, вспомним, что сведение задаётся парой процедур f и h (см. диаграмму на с. 242). Если (f_{AB}, h_{AB}) и (f_{BC}, h_{BC}) задают сведения $A \to B$ и $B \to C$, то сведение от A к C задаётся композициями этих процедур: $f_{BC} \circ f_{AB}$ переводит вход задачи A во вход задачи C, а $h_{AB} \circ h_{BC}$ по решению для задачи C строит решение для задачи A.

Таким образом, если мы доказали NP-полноту задачи B (то есть установили, что к ней сводится любая NP-задача A), а также свели B к C, то и задача C будет NP-полной (любая NP-задача A сводится к C через B).

> Теперь мы готовы определить класс самых трудных задач поиска.
>
> Задача поиска называется NP-полной, если любая задача поиска к ней сводится.

На первый взгляд непонятно, как вообще можно доказать, что любая мыслимая NP-задача сводится к данной конкретной задаче. Тем не менее это возможно, и в левом столбце нашей таблицы представлены наиболее известные такие примеры NP-полных задач. В разделе 8.3 мы увидим, как эти задачи сводятся друг к другу и почему любые задачи поиска сводятся к ним.

Разложение на множители

Мы привели много примеров задач из класса P и NP-полных задач, но бывают и промежуточные ситуации. В начале книги мы говорили об известной трудной задаче факторизации (разложении на множители: требуется представить число как произведение простых). Хотя мы не знаем полиномиального алгоритма для этой задачи, не похоже, чтобы задача факторизации была NP-полной. (Можно доказать, что она входит в класс NP.) В отличие от типичных NP-задач, она всегда имеет решение; другое отличие (возможно, связанное с предыдущим) заключается в том, что эта задача может быть эффективно решена на (пока воображаемом) квантовом компьютере (см. главу 10), а про NP-полные задачи это неизвестно (и, похоже, невозможно). Так или иначе, для разложения на множители не известно ни полиномиального алгоритма, ни доказательства NP-полноты.

8.3. Сведе́ния

Мы докажем NP-полноту задач на рис. 8.7, начиная с выполнимости, сводя их друг к другу по стрелкам.

Рис. 8.7. Сводимость для задач поиска.

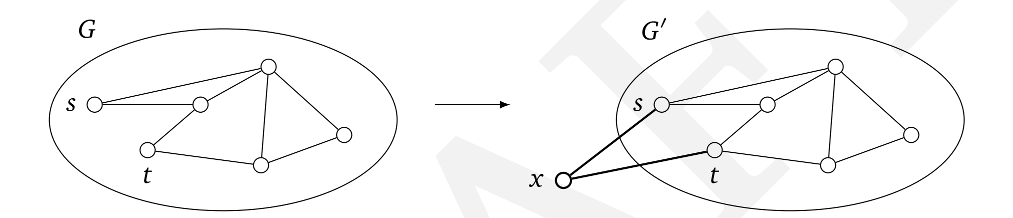
все задачи из NP
↓
выполнимость
↓
3-выполнимость
↙ ↘
независимое множество — трёхдольное сочетание
независимое множество → вершинное покрытие, клика
трёхдольное сочетание → УНЕ
УНЕ → сумма подмножества, ЦЛП, гамильтонов цикл
гамильтонов цикл → коммивояжёр

Для тренировки сначала покажем, что два варианта задачи Гамильтона сводятся друг к другу.

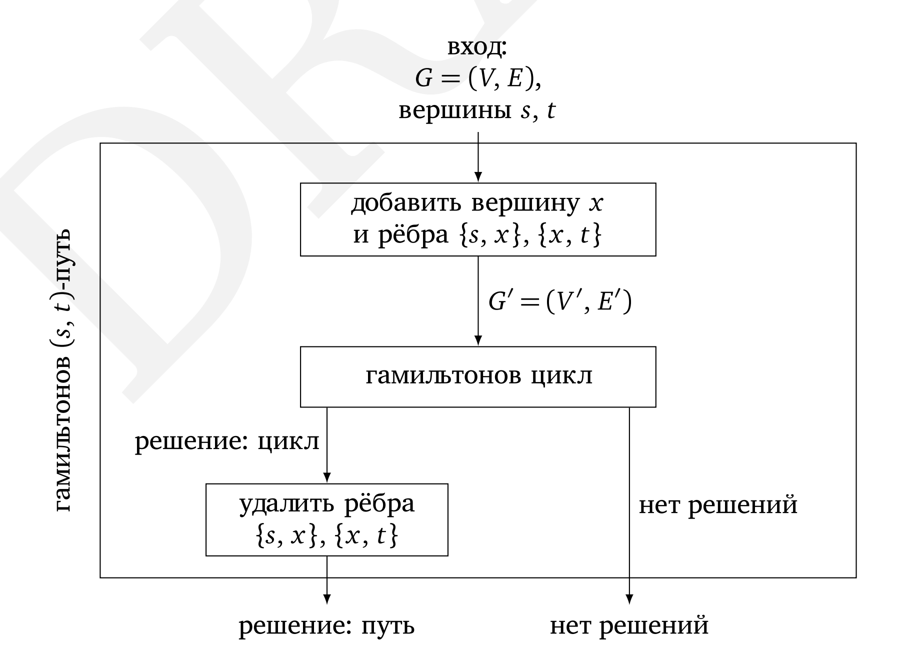
Гамильтонов (s, t)-путь → гамильтонов цикл

Напомним, что в задаче о гамильтоновом цикле дан граф и необходимо найти цикл, проходящий через каждую вершину по одному разу. Можно также рассмотреть похожую задачу о гамильтоновом (s, t)-пути, в которой дан граф и две его вершины s и t, а найти необходимо путь из s в t, проходящий через все вершины по одному разу. Может ли оказаться, что цикл искать проще, чем (s, t)-путь? Сведя вторую задачу к первой, мы покажем, что нет.

Сведение преобразует вход $(G = (V, E), s, t)$ задачи о гамильтоновом (s, t)-пути во вход $G' = (V', E')$ задачи о гамильтоновом цикле: G' получается из G добавлением новой вершины x и двух рёбер, соединяющих x с s и t.



Другими словами, $V' = V \cup \{x\}$ и $E' = E \cup \{\{s, x\}, \{x, t\}\}$. Найдя гамильтонов *цикл* в графе G', легко получить из него гамильтонов (s, t)-*путь* в G: цикл обязан включать в себя рёбра $\{s, x\}$ и $\{x, t\}$, и достаточно их удалить.



Почему это действительно сведение? Разберём два случая.

1. Гамильтонов цикл удалось найти.

Поскольку соседями добавленной вершины x являются только s и t, любой гамильтонов цикл в G' обязан последовательно проходить рёбра $\{t, x\}$ и $\{x, s\}$ (в ту или другую сторону). Тогда оставшаяся часть цикла — это путь из s в t.

2. В построенном при сведении графе нет гамильтонова цикла.

Нужно показать, что решений нет и у исходной задачи о гамильтоновом (s, t)-пути. Рассуждая от противного, мы должны предположить, что в графе G есть гамильтонов (s, t)-путь, и показать, что в G' есть гамильтонов цикл. Но это очевидно: добавив рёбра $\{t, x\}$ и $\{x, s\}$ к гамильтонову пути, получим гамильтонов цикл.

Надо также не забыть проверить, что процедуры пред- и постобработки, которые задают сведение, имеют полиномиальное время работы. (В данном случае это очевидно.)

Несложно также построить и обратное сведение — гамильтонова цикла к гамильтонову (s, t)-пути. Вместе они показывают, что два рассмотренных варианта задачи Гамильтона имеют почти одинаковую сложность. В данном случае это не так удивительно (их формулировки очень близки), но вскоре мы построим сведения между внешне совсем разными задачами.

3-ВЫПОЛНИМОСТЬ → НЕЗАВИСИМОЕ МНОЖЕСТВО

Эти задачи на вид совершенно разные. В задаче 3-выполнимости на вход даётся список дизъюнктов, каждый из которых содержит не более трёх литералов, например

$$(\bar{x} \vee y \vee \bar{z})\,(x \vee \bar{y} \vee z)\,(x \vee y \vee z)\,(\bar{x} \vee \bar{y}).$$

Требуется найти выполняющий набор (значения переменных, при которых все дизъюнкты истинны). В задаче о независимом множестве на вход даётся граф и число g и необходимо найти g вершин, никакие две из которых не соединены ребром. Но как связать булеву логику с графами?

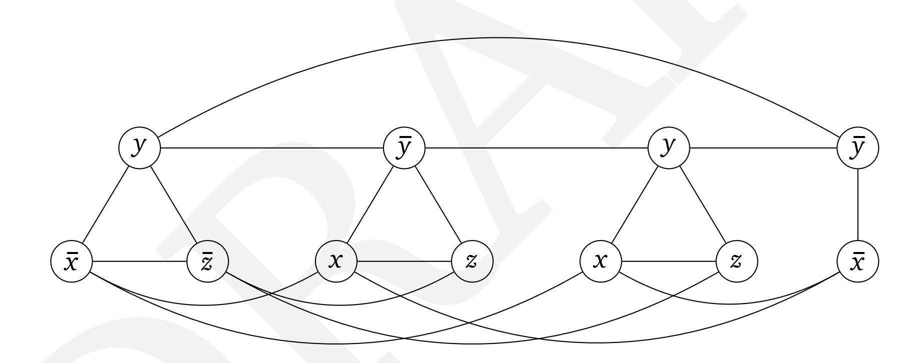
Давайте подумаем. Чтобы построить выполняющий набор, мы должны выбрать по одному литералу в каждом дизъюнкте и присвоить ему значение `true`. Но выбирать нужно согласованно: если в каком-то дизъюнкте мы выбрали $\bar{x}$, мы уже не можем выбрать x в другом. Любой такой согласованный набор литералов задаёт выполняющий набор (переменным, для которых ни один из двух соответствующих литералов не был выбран, можно присвоить произвольные значения). Обратно, если есть выполняющий набор, то можно выбрать в каждом дизъюнкте истинный литерал (любой, если их несколько) и получить согласованный набор литералов. Мы свели задачу к поиску согласованного набора литералов (по одному на дизъюнкт).

Как это сформулировать в терминах независимых множеств? Для каждого дизъюнкта вида $(x \vee \bar{y} \vee z)$ нарисуем треугольник, вершины которого пометим x, $\bar{y}$, z. В треугольнике каждая пара вершин соединена ребром, и значит, в независимом множестве может оказаться не более одной из них. Для дизъ-

юнкта из двух литералов нарисуем отрезок, соединяющий вершины с такими пометками. (Дизъюнкты с одной переменной можно не рассматривать: они определяют значение этой переменной и его можно подставить в остальные дизъюнкты, упростив формулу.) В полученном графе независимое множество может содержать не более одной вершины из каждой группы (дизъюнкта). Чтобы выбрать из каждой группы ровно одну вершину, потребуем найти столько независимых вершин, сколько есть дизъюнктов (в нашем примере — 4).

Пока, однако, ничего не мешает выбирать несогласованные литералы. Чтобы запретить такой выбор, соединим в графе ребром каждую пару противоположных литералов. Граф, соответствующий нашему примеру, показан на рисунке 8.8.

Рис. 8.8. Граф для формулы $(\bar{x} \vee y \vee \bar{z})\,(x \vee \bar{y} \vee z)\,(x \vee y \vee z)\,(\bar{x} \vee \bar{y})$.
1. Для каждого дизъюнкта из трёх литералов граф G содержит треугольник, вершины которого помечены литералами данного дизъюнкта. Дизъюнктам длины два соответствуют просто рёбра. В графе также проведены дополнительные рёбра, соединяющие все пары противоположных литералов.
2. Необходимо найти независимое множество размера хотя бы $g = 4$.



Легко видеть, что время построения такого графа полиномиально. Но сведение требует не только эффективного способа перевода входов первой задачи во входы второй (функция f в диаграмме на с. 242), но и эффективного способа восстановления решения для первой задачи по решению для второй (функция h). По существу это уже описано выше, но скажем подробнее. Нам надо убедиться в следующем:

1. По независимому множеству S, состоящему из g вершин графа G, легко построить выполняющий набор для I.

Множество S не может содержать пару вершин, помеченных литералами x и $\bar{x}$, поскольку любые две такие вершины соединены ребром. Если в S есть вершина, помеченная x, присвоим переменной x значение `true`; если в S есть вершина с пометкой $\bar{x}$, то присвоим x значение `false`; наконец, если нет ни таких, ни сяких, присвоим x произвольное значение. Поскольку S содержит

ровно g вершин, в каждом дизъюнкте выбрана одна из вершин, и дизъюнкт будет выполнен благодаря соответствующей переменной.

2. Если в G нет независимого множества размера g, то формула I невыполнима.

Вновь рассуждая от противного, предположим, что I выполнима, и покажем, что тогда в G найдётся независимое множество размера g. Возьмём в каждом дизъюнкте выполненный литерал (любой, если их несколько) и добавим соответствующую вершину в S. (Понятно ли, почему полученное множество S будет независимым?)

Выполнимость → 3-выполнимость

Здесь, как часто бывает, мы сводим задачу к её собственному *частному случаю*. Такое сведение показывает, что задача остаётся трудной, даже если мы некоторым образом ограничиваем возможные входы — в данном случае запрещаем дизъюнктам содержать более трёх литералов. Чтобы построить сведение, нам нужно как-то избавиться от дизъюнктов из четырёх и более литералов, при этом не изменив задачу по существу. Это делается с помощью такого трюка: длинный дизъюнкт $(a_1 \vee a_2 \vee \ldots \vee a_k)$ можно заменить на два более коротких $(a_1 \vee a_2 \vee y)(\bar{y} \vee a_3 \vee \ldots \vee a_k)$, где y — новая переменная, значение которой можно подобрать в том и только том случае, когда разбиваемый дизъюнкт был истинным. Повторяя этот приём, получим множество дизъюнктов

$$(a_1 \vee a_2 \vee y_1)\,(\bar{y}_1 \vee a_3 \vee y_2)\,(\bar{y}_2 \vee a_4 \vee y_3)\,\ldots(\bar{y}_{k-3} \vee a_{k-1} \vee a_k),$$

где y_i — это новые переменные. Обозначим полученную формулу (в 3-КНФ) через I'. Легко видеть, что I' строится по I за полиномиальное время.

Почему же данное сведение корректно? Чтобы понять, что I выполнима тогда и только тогда, когда I' выполнима, заметим, что для любого набора значений для переменных a_i

$$\left\{\begin{array}{c}(a_1 \vee a_2 \vee \ldots \vee a_k)\\ \text{выполнен}\end{array}\right\} \Longleftrightarrow \left\{\begin{array}{c}\text{найдётся набор значений } y_i,\\ \text{для которого все дизъюнкты}\\ (a_1 \vee a_2 \vee y_1)\ldots(\bar{y}_{k-3} \vee a_{k-1} \vee a_k)\\ \text{выполнены}\end{array}\right\}.$$

Действительно, пусть все дизъюнкты справа выполнены. Тогда хотя бы один из литералов $a_1, \ldots, a_k$ должен быть выполнен: иначе должен быть выполнен y_1, за ним — y_2 и все остальные y_i, что привело бы к тому, что последний дизъюнкт не выполнен. А значит, выполнен и дизъюнкт $(a_1 \vee a_2 \vee \ldots \vee a_k)$.

Обратно, пусть в дизъюнкте $(a_1 \vee a_2 \vee \ldots \vee a_k)$ выполнен литерал a_i. Присвоим тогда переменным $y_1, \ldots, y_{i-2}$ значение `true`, а оставшимся — значение `false`. Легко видеть, что при этом все дизъюнкты справа будут выполнены.

Таким образом, любой вход задачи выполнимости можно преобразовать во вход задачи 3-выполнимости. Искомое сведение построено, и тем самым мы показали, что частный случай 3-выполнимости не легче общей задачи выполнимости.

На самом деле верно даже большее: задача 3-выполнимости остаётся трудной, даже если мы также потребуем, чтобы каждая переменная входила в не более чем три дизъюнкта. Чтобы показать это, нам надо научиться как-то избавляться от переменных, которых входят в формулу больше трёх раз.

Пусть какая-то переменная x встречается $k > 3$ раз. Заменим первое её вхождение на новую переменную x_1, второе — на новую переменную x_2 и так далее. Теперь надо добавить ограничения, обязывающие все эти переменные быть равными друг другу. Сделаем это, добавив дизъюнкты

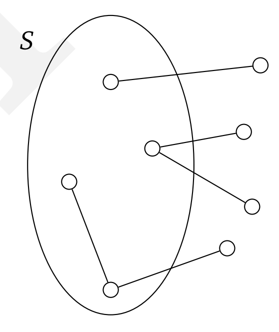
$$(\bar{x}_1 \vee x_2)\,(\bar{x}_2 \vee x_3)\ldots(\bar{x}_k \vee x_1).$$

Заменив подобным образом все частые переменные, мы получим формулу, в которую каждая переменная входит не более трёх раз (а каждый литерал — не более двух раз).

Независимое множество → вершинное покрытие

Бывают хитрые сведения, которые связывают совсем не похожие друг на друга задачи. Но иногда речь идёт по существу о двух формулировках одной и той же задачи, и сведение это констатирует. Вот пример такого рода. Чтобы свести задачу о независимом множестве к задаче о вершинном покрытии, заметим, что множество вершин S является вершинным покрытием графа $G = (V, E)$ (то есть затрагивает каждое ребро графа) тогда и только тогда, когда множество вершин $V - S$ независимо в G (рис. 8.9).

Рис. 8.9. S является вершинным покрытием тогда и только тогда, когда $V - S$ является независимым множеством: «любое ребро имеет конец в S» означает, что «никакое ребро не может иметь оба конца вне S», то есть «никакие две вершины $V - S$ не соединены ребром».



Значит, для решения задачи о независимом множестве на входе (G, g) достаточно решить задачу о вершинном покрытии на входе $(G, |V| - g)$. Если такое вершинное покрытие найдётся, то в качестве независимого множества нужно взять все остальные вершины. В противном случае в G нет независимого множества размера g.

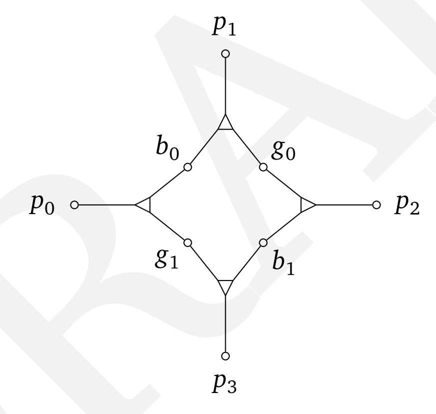
Независимое множество → клика

Эти две задачи сводятся друг к другу столь же просто. Назовём *дополнением* (complement) графа $G = (V, E)$ граф $\overline{G} = (V, \overline{E})$, где $\overline{E}$ — отсутствующие в E рёбра (соединяющие пары вершин, не соединённые в G). Множество вершин S будет независимым в G, если и только если S является кликой в $\overline{G}$: отсутствие ребра в G означает его наличие в $\overline{G}$. Таким образом, задачу о независимом множестве можно свести к задаче о клике, преобразовав вход (G, g) в $(\overline{G}, g)$.

3-выполнимость → трёхдольное сочетание

Вновь нам нужно свести задачу 3-выполнимости к совсем другой на вид задаче: среди заданных троек (мальчик, девочка, домашнее животное) надо найти такое подмножество, которое покрывает каждого участника по одному разу. Для сведения надо имитировать поведение булевых переменных и функций с помощью подходящих троек.

Рассмотрим сначала конструкцию из четырёх троек, каждая из которых представлена треугольником и соединяет мальчика, девочку и животное (на рисунке использованы буквы b, g, p — от английских слов boy, girl, pet).



Допустим, что мальчики b_0 и b_1, а также девочки g_0 и g_1 не входят больше ни в какие тройки. (Кто-то из p_1, p_2, p_3, p_4 должен входить в другие тройки, иначе у задачи по очевидным причинам просто не было бы решения.) Видно, что для охвата всех мальчиков и девочек есть ровно два варианта: либо надо взять пару троек (b_0, g_1, p_0), (b_1, g_0, p_2) (а две другие не брать), либо же пару троек (b_0, g_0, p_1), (b_1, g_1, p_3). Значит, данный блок (по-английски в таких случаях говорят «gadget») может находиться в одном из двух состояний — прямо как булева переменная. В каждом из этих состояний остаются свободными два животных (стоящих напротив друг друга): либо p_0 и p_2, либо p_1 и p_3.

Чтобы переделать вход задачи 3-выполнимости во вход задачи о трёхдольном сочетании, мы первым делом заводим по такой конструкции для каждой переменной x. Обозначим входящие в эту конструкцию переменные через p_{x1}, b_{x0}, g_{x1} и так далее. Договоримся, что $x = \texttt{true}$, если мальчик b_{x0} попал в тройку с девочкой g_{x1}, и $x = \texttt{false}$, если с девочкой g_{x0}.

Теперь нам предстоит создать тройки, которые будут моделировать дизъюнкты. Для дизъюнкта $c = (x \vee \bar{y} \vee z)$ заведём мальчика b_c и девочку g_c. Каждый из них будет входить в три тройки, соответствующие трём литералам нашего дизъюнкта. Эти три тройки должны отражать три способа выполнить дизъюнкт: (1) $x = \texttt{true}$, (2) $y = \texttt{false}$, (3) $z = \texttt{true}$. Для (1) возьмём тройку (b_c, g_c, p_{x1}), где p_{x1} — это p_1 в конструкции для x. Эту тройку можно использовать, если p_1 остался свободным, то есть если $x = \texttt{true}$. Аналогичным образом добавим тройки для литералов $\bar{y}$ и z (одну из троек (b_c, g_c, p_{y0}) или (b_c, g_c, p_{y2}), а также одну из троек (b_c, g_c, p_{z1}) или (b_c, g_c, p_{z3})). Возможность подобрать домашнее животное для b_c и g_c означает, что дизъюнкт c выполнен.

Тут есть проблема: если один и тот же литерал встречается в нескольких дизъюнктах, то может не хватить животных: для каждой переменной их только два (для каждого из двух её возможных значений). Но мы уже знаем, как переделать формулу в такую, где каждый литерал встречается не более двух раз, и тогда их хватит, потому что в каждом блоке есть два животных для переменной и два для её отрицания.

Кажется, что всё готово. По решению полученной задачи о трёхдольном сочетании легко восстановить выполняющий набор для исходной формулы: для каждой переменной надо посмотреть на изображающий её блок. Напротив, чтобы по выполняющему набору построить трёхдольное сочетание, закодируем значения переменных на языке блоков, то есть возьмём тройки (b_{x0}, g_{x1}, p_{x0}) и (b_{x1}, g_{x0}, p_{x2}), если $x = \texttt{true}$, и тройки (b_{x0}, g_{x0}, p_{x1}) и (b_{x1}, g_{x1}, p_{x3}), если $x = \texttt{false}$. Кроме того, для каждого дизъюнкта c выберем в нём истинный литерал и возьмём соответствующую ему тройку (содержащую b_c, g_c и животное, оставшееся свободным при выбранном значении переменной).

Осталась одна небольшая тонкость: в построенном трёхдольном сочетании *могли остаться беспризорные животные*, и даже понятно сколько: если в формуле n переменных и m дизъюнктов, то $2n - m$ питомцев не попадут в тройки (тут всё сходится: это число неотрицательно, поскольку каждая переменная входит в формулу не более трёх раз, а каждый дизъюнкт содержит не менее двух литералов). Понятно, как тут быть: добавим $2n - m$ пар мальчик-девочка, которым годятся любые животные (то есть добавим все тройки, включающие одну из таких пар и одно животное).

Трёхдольное сочетание → уравнения в нулях и единицах

Напомним, что входом задачи УНЕ является битовая матрица $\mathbf{A}$ размера $m \times n$, а найти нужно битовый вектор $\mathbf{x} = (x_1, \ldots, x_n)$, удовлетворяющий уравнению

$$\mathbf{Ax} = \mathbf{1},$$

где через $\mathbf{1}$ обозначен вектор из единиц. Как же сформулировать задачу о трёхдольном сочетании в этих терминах?

Это несложно: искомый объект (в данном случае трёхдольное сочетание) описывается переменными, а уравнения задают ограничения. Как это сде-

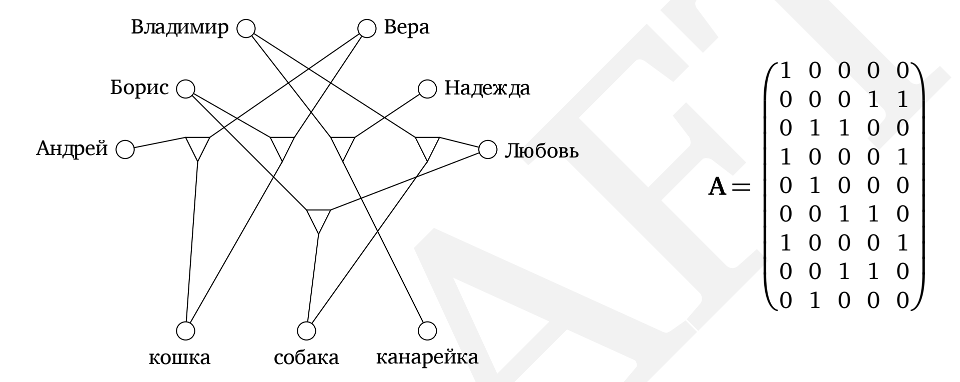
лать в нашем случае? Пусть у нас m мальчиков, m девочек, m животных, а также n троек. Заведём n булевых переменных $x_1, \ldots, x_n$, по одной для каждой тройки; $x_i = 1$ означает, что i-я тройка вошла в сочетание.

Теперь нужно записать уравнения, гарантирующие, что решение, задаваемое переменными $x_1, \ldots, x_n$, является корректным сочетанием. Рассмотрим какого-нибудь мальчика. Пусть он входит в тройки с номерами $j_1, \ldots, j_k$. Нам надо записать, что выбрана ровно одна из этих троек:

$$x_{j_1} + x_{j_2} + \ldots + x_{j_k} = 1.$$

Такие уравнения нужно записать для всех мальчиков, девочек и животных.

Для нашего примера получится такая матрица $\mathbf{A}$:



$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Столбцы матрицы $\mathbf{A}$ соответствуют пяти представленным тройкам, а строки соответствуют Андрею, Борису, Владимиру, Вере, Надежде, Любови, кошке, собаке и канарейке.

Убедиться в том, что решения для этих задач переделываются одно в другое, не представляет никакого труда.

УНЕ → СУММА ПОДМНОЖЕСТВА

Данное сведение связывает два частных случая задачи целочисленного линейного программирования: в обоих случаях переменные булевы, но в одном случае много уравнений с коэффициентами 0 и 1, а в другом — всего одно уравнение, но зато с произвольными целочисленными коэффициентами. Идея сведения простая, но важная: целые числа можно задавать с помощью битовых векторов.

Для примера рассмотрим такую матрицу:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Мы ищем столбцы матрицы $\mathbf{A}$, которые при сложении дают вектор из одних единиц. Столбцы матрицы можно интерпретировать как числа в двоич-

ной системе счисления (младшие разряды снизу), тогда задача преобразуется в такую: нужно выбрать некоторые из чисел 18, 5, 4, 8, получив сумму $11111_2 = 31$. Таким образом, мы построили сведение задачи УНЕ к задаче о сумме подмножества.

За исключением одной детали — мы забыли про *переносы*. Из-за переносов сумма пятибитовых чисел может оказаться равной 31, даже когда сумма соответствующих векторов не равна (1, 1, 1, 1, 1): например, $5 + 6 + 20 = 00101_2 + 00110_2 + 10100_2 = 11111_2 = 31$. Но побороть эту проблему в нашем случае просто: будем рассматривать столбцы матрицы как записи чисел не в двоичной системе счисления, а в системе по основанию $(n + 1)$. Тогда переносов просто не будет (поскольку слагаемых не больше n, а все цифры — нули или единицы).

УНЕ → ЦЛП

Если одна задача является частным случаем другой, то сведение (частной задачи к общей) тривиально: обе функции f и h (см. диаграмму сведения на с. 242) тождественны. Скажем, задача 3-выполнимости является частным случаем задачи выполнимости. Это простое замечание часто оказывается полезным: скажем, задача о покрытии множествами NP-полна, потому что она обобщает задачу о вершинном покрытии (а заодно и задачу о трёхдольном сочетании). В упражнении 8.10 приведено ещё несколько подобных примеров.

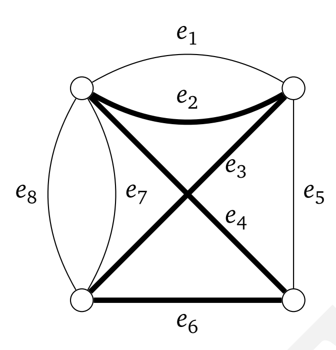
То же самое (с точностью до небольшой переформулировки) относится к сведению УНЕ к ЦЛП. В задаче ЦЛП мы хотим найти целочисленный вектор $\mathbf{x}$, для которого $\mathbf{Ax} \leqslant \mathbf{b}$. Чтобы записать в такой форме вход задачи УНЕ (систему уравнений с булевыми переменными), нужно всего лишь переписать каждое равенство как два неравенства (как мы делали в разделе 7.1.4) и ещё для каждой переменной x_i добавить два неравенства $x_i \leqslant 1$ и $-x_i \leqslant 0$.

УНЕ → гамильтонов цикл

В задаче о гамильтоновом цикле мы ищем в графе цикл, который проходит через каждую вершину ровно один раз. Мы докажем NP-полноту этой задачи в два приёма: сначала мы сведём УНЕ к чуть более общей задаче о *гамильтоновом цикле с парными рёбрами*, после чего поймём, как свести это обобщение к исходной задаче о гамильтоновом цикле.

В задаче о гамильтоновом цикле с парными рёбрами дан граф $G = (V, E)$ и множество $C \subseteq E \times E$ пар рёбер. Необходимо найти цикл, который: (1) как и в исходной задаче, проходит через каждую вершину ровно один раз и (2) для каждой пары рёбер $(e, e') \in C$ проходит *ровно по одному* из этих рёбер. На рис. 8.10 показан пример такой задачи; решение выделено жирными линиями. Заметьте, что теперь мы разрешаем кратные рёбра (несколько рёбер, соединяющих одни и те же вершины). Во многих задачах на графах смысла в таких дублирующих рёбрах нет, но в рассматриваемой нами сейчас задаче это осмысленно, потому что в разные пары могут входить разные копии одного и того же ребра.

Рис. 8.10. Задача о гамильтоновом цикле в графе с парными рёбрами (множество C равно $\{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$) и её решение.



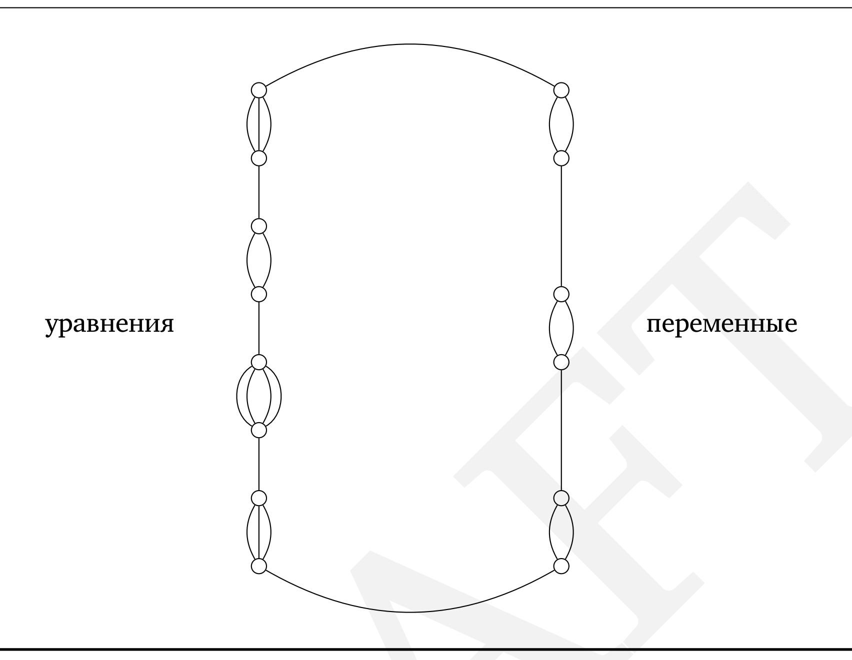
Итак, начнём сводить УНЕ к гамильтонову циклу с парными рёбрами. Имея систему уравнений $\mathbf{Ax} = \mathbf{1}$ (где $\mathbf{A}$ — это матрица размера $m \times n$ из нулей и единиц, задающая m уравнений от n переменных) мы построим (см. рис. 8.11) цикл, соединяющий $m + n$ семейств кратных рёбер. Для каждой переменной x_i в нём два кратных ребра (соответствующих случаям $x_i = 0$ и $x_i = 1$). Для каждого уравнения $x_{j_1} + \ldots + x_{j_k} = 1$ с k переменными будет коллекция из k кратных рёбер, по одному для каждой переменной. Ясно, что гамильтонов цикл в таком графе должен «выбрать» значение для каждой переменной и «выбрать» по одной переменной в каждом уравнения.

Конечно, это ещё не всё. Мы пока отразили в графе только размеры матрицы $\mathbf{A}$, но не её структуру — легко догадаться, что для этого мы используем возможность указывать пары рёбер. Для каждого уравнения (напомним, что их m штук) и каждой переменной x_i из этого уравнения добавим в C пару (e, e'), где e — ребро, соответствующее вхождению этой переменной в это уравнение (слева на рисунке 8.11), а e' — ребро, соответствующее случаю $x_i = 0$ (на рисунке справа). Этим мы вынуждаем ту переменную в уравнении, через которую прошёл цикл, быть истинной, а остальные быть ложными. На этом конструкция завершена.

Почему это работает? Рассмотрим любое решение полученной задачи о гамильтоновом цикле с парными рёбрами. Как мы уже говорили, такое решение выбирает значение для каждой переменной, а также выбирает по одной переменной в каждом уравнения. Проверим, что выбранные таким образом значения удовлетворяют всем уравнениям. Условия на пары гарантируют, что те переменные, через которые цикл не проходит в левой половине, равны 0, а те, через которые цикл проходит, равны 1. Поэтому в каждом уравнении есть ровно одна единица, так что оно выполняется. Легко выполнить и обратное построение (гамильтонова пути по решению системы уравнений).

Как избавиться от парных рёбер. Осталось свести задачу о гамильтоновом цикле с парными рёбрами к обычной задаче о гамильтоновом цикле, то есть как-то избавиться от парных рёбер (сделать множество C пустым).

Рис. 8.11. Сведение УНЕ к гамильтонову циклу с парными рёбрами.

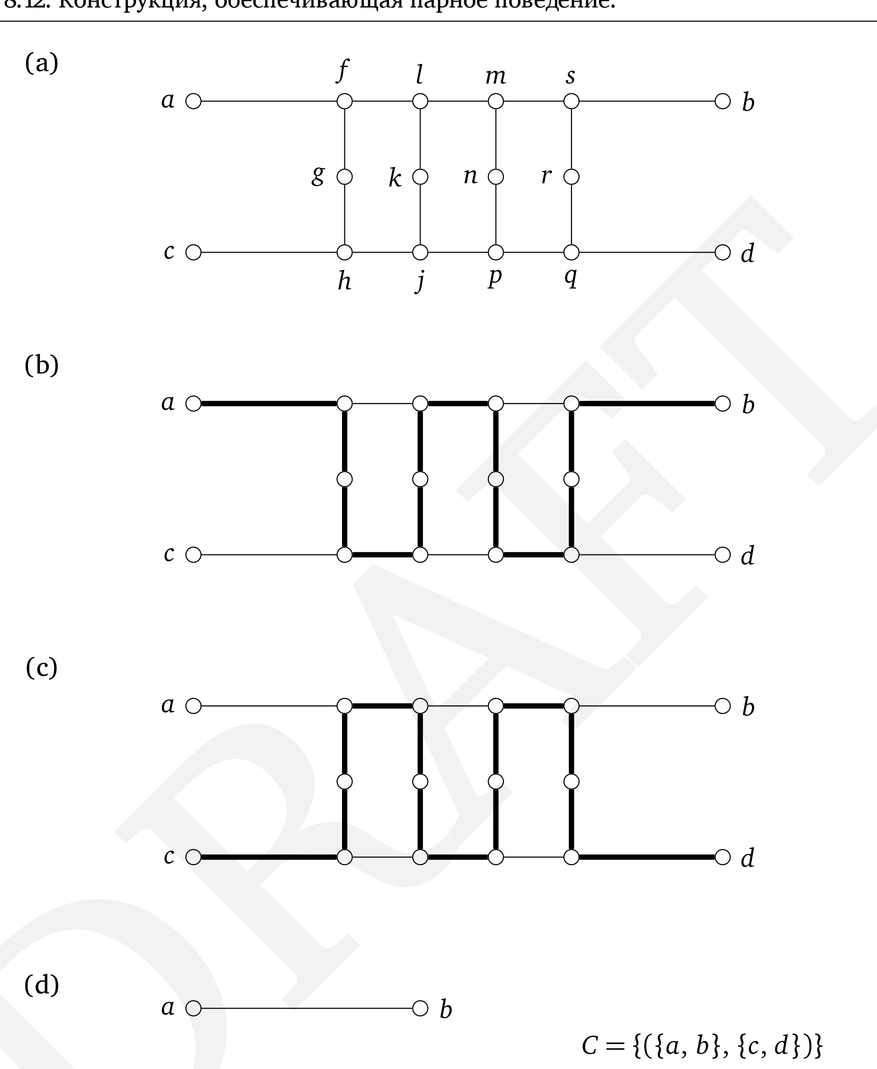


Это делается с помощью блоков, изображённых на рис. 8.12. Пусть этот блок является частью какого-то графа G, соединённой с другими вершинами лишь в точках a, b, c, d. Давайте покажем, что гамильтонов цикл в графе G может проходить блок лишь двумя способами, изображёнными жирными линиями. Ясно, что вертикальные линии неизбежны, иначе мы не посетим средний ряд вершин. Посмотрим на одну из них (не крайнюю): пройдя по ней, цикл должен повернуть налево или направо, и из рисунка видно, что этот выбор определяет всё дальнейшее (и всё предшествующее) поведение: мы попадаем на следующую вертикальную линию, которая продолжает цикл, и следующий выбор однозначен (иначе получится замкнутый круг).

Таким образом, эта конструкция ведёт себя точно так же, как два ребра $\{a, b\}$ и $\{c, d\}$, объединённые в пару (рис. 8.12(d)), в варианте задачи с парными рёбрами.

Теперь понятно, как свести задачу о гамильтоновом цикле с парными рёбрами к задаче об обычном гамильтоновом цикле. Будем постепенно уменьшать количество пар, пока они все не исчезнут. Взяв какую-то пару рёбер $(\{a, b\}, \{c, d\}) \in C$, мы заменяем эту пару на конструкцию рис. 8.12(a). Но надо иметь в виду, что рёбра $\{a, b\}$ или $\{c, d\}$ могли входить и в другие пары. Ничего страшного: во всех оставшихся парах, содержащих ребро $\{a, b\}$, заменяем его на ребро $\{a, f\}$ из только что добавленной конструкции (прохождение по нему кодирует прохождение по ребру $\{a, b\}$ исходного графа). Аналогично заменяем $\{c, d\}$ (в других парах) на $\{c, h\}$. Такое преобразование графа полиномиально, поскольку на каждую пару рёбер добавляется по

Рис. 8.12. Конструкция, обеспечивающая парное поведение.



12 вершин. Легко видеть, что гамильтоновы циклы в полученном графе находятся во взаимно однозначном соответствии с гамильтоновыми циклами в исходном графе, согласованными с множеством пар рёбер C.

Гамильтонов цикл → коммивояжёр

По данному графу $G = (V, E)$ построим следующий граф для задачи коммивояжёра: множество городов совпадает с $|V|$; расстояние между городами u и v положим равным 1, если $\{u, v\} \in E$, и $1 + \alpha$ в противном случае. Значение

$\alpha \geqslant 1$ мы подберём позже. В полученном графе будем искать цикл суммарного веса не более $|V|$.

Ясно, что если в исходном графе есть гамильтонов цикл, то и в полученном графе есть гамильтонов цикл веса ровно $|V|$. Если же гамильтонова цикла в исходном графе нет, то любой гамильтонов цикл в полученном графе будет иметь вес хотя бы $|V|+\alpha$ (ему придётся хотя бы раз пройти по ребру веса $1+\alpha$, а все остальные $|V|-1$ рёбер имеют вес хотя бы 1). Итак, мы свели задачу о гамильтоновом цикле к задаче коммивояжёра.

Зачем, однако, мы ввели параметр α? Ведь для сведения достаточно было бы взять, скажем, $\alpha=1$. При $\alpha=1$ веса рёбер в построенном графе удовлетворяют неравенству треугольника: для городов i, j, k выполняется неравенство $d_{ij}+d_{jk} \geqslant d_{ik}$ (действительно, $d_{ij}+d_{jk} \geqslant 1+1=2 \geqslant d_{ik}$). Таким образом, мы свели задачу о гамильтоновом цикле к задаче коммивояжёра с дополнительным ограничением (расстояния удовлетворяют неравенству треугольника). Этот частный случай часто возникает на практике и в некотором смысле проще общего случая: для него есть эффективные *приближённые* алгоритмы.

Если же взять большее значение α, то веса рёбер уже не будут удовлетворять неравенству треугольника, но зато наше сведение будет обладать другим интересным свойством. А именно, в полученном графе либо есть гамильтонов цикл веса $|V|$, либо любой гамильтонов цикл имеет вес как минимум $|V|+\alpha$, и это число может быть сильно больше $|V|$. В главе 9 мы увидим, что такой зазор позволяет показать, что (в предположении $\mathrm{P} \neq \mathrm{NP}$) для общей задачи коммивояжёра нет эффективных приближённых алгоритмов.

Любая задача класса NP → выполнимость

Мы свели задачу о выполнимости к разным задачам из схемы на рис. 8.7. Сейчас мы замкнём круг и убедимся, что верно и обратное, установив, что все они — и вообще любые задачи поиска (= задачи из NP) — сводятся к задаче выполнимости. Тем самым мы докажем, что все задачи на рис. 8.7 эквивалентны друг другу и NP-полны.

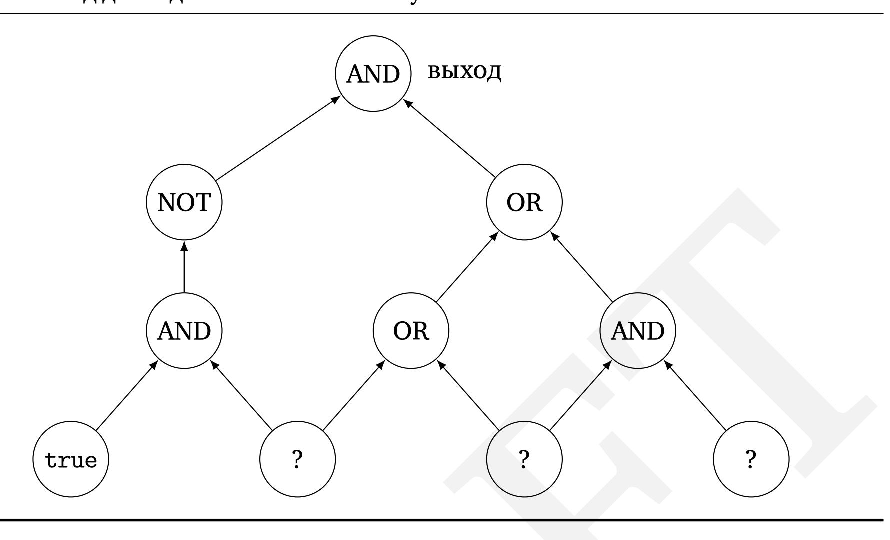
Мы сведём произвольную задачу из NP к чуть более общему варианту задачи о выполнимости — задаче выполнимости булевой схемы (circuit SAT). Входом данной задачи является *булева схема* (см. рис. 8.13 и раздел 7.7), то есть ациклический ориентированный граф, в котором есть вершины следующих типов:

- Элементы AND и OR, имеющие входящую степень 2.
- Элементы NOT, имеющие входящую степень 1.
- *Известные входы* (входящая степень ноль), помеченные константой `true` или `false`.
- *Неизвестные входы* (входящая степень 0), помеченные знаком «?».

Одна из вершин схемы имеет выходную степень 0 и помечена как «выход».

Выбрав значения для неизвестных входов, мы можем определить значение в каждой вершине (в том числе и в выходной вершине), пользуясь законами булевой логики (типа `false` ∨ `true` = `true`). Например, при подстановке значений `true`, `false`, `true` схема рис. 8.13 вычисляет значение `false`.

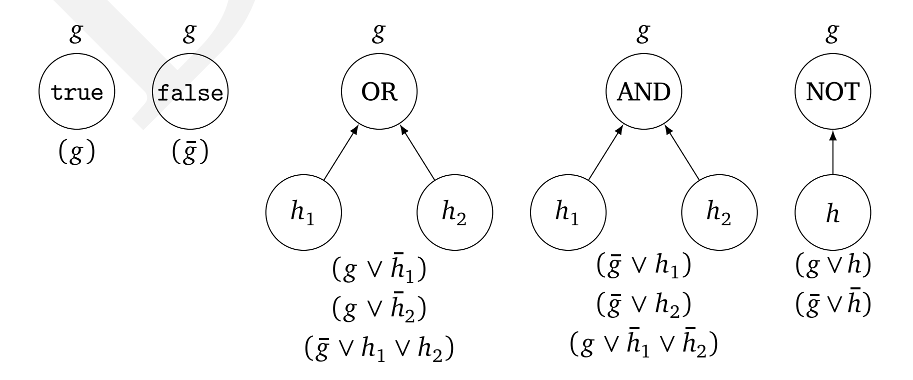
Рис. 8.13. Вход для задачи выполнимости булевой схемы.



Задача о выполнимости булевой схемы возникает, когда задана булева схема, а найти нужно набор значений неизвестных, при котором на выходе схемы будет `true` (или же сообщить, что такого набора нет). Например, набор (`false`, `true`, `true`) выполняет схему рис. 8.13.

Эта задача обобщает рассмотренную раньше задачу о выполнимости, ведь формулу в КНФ легко задать схемой. Такая схема будет вычислять конъюнкцию дизъюнктов (с помощью нескольких элементов типа AND), а каждый дизъюнкт будет дизъюнкцией литералов. Наконец, литерал — это либо неизвестный вход, либо его отрицание. (Известные входы при этом не нужны.)

Это обобщение не является принципиальным: сейчас мы сведём задачу о выполнимости булевой схемы к задаче выполнимости, заменив схему на формулу в КНФ. В ней появятся новые переменные (кроме входных неизвестных): для каждой вершины заведём новую переменную g и запишем следующие дизъюнкты, гарантирующие, что значение этой переменной действительно вычисляется по правилам логики:



(Понятно ли, почему достаточно записать такие дизъюнкты?) Чтобы заставить схему вычислять значение `true`, добавим также дизъюнкт (g) для выходного элемента g. Выполняющие наборы для полученной формулы в КНФ находятся во взаимно однозначном соответствии с выполняющими наборами исходной схемы.

Вернёмся теперь к нашей основной задаче — сведению произвольной задачи поиска A к задаче выполнимости булевой схемы. Как построить такое сведение? Кажется, это будет непросто, *ведь мы ничего не знаем об A, кроме того, что она является задачей поиска.*

Значит, воспользоваться нужно именно этим. Как мы помним, задача поиска задаётся проверяющим алгоритмом. Именно, для A есть алгоритм $\mathscr{C}$, проверяющий по входу I и кандидату S, действительно ли S является решением. Более того, этот алгоритм осуществляет такую проверку за полиномиальное от длины I время (из чего, в частности, следует, что длина S должна быть ограничена полиномом от длины I, иначе мы не успеем даже прочесть S).

Вспомним теперь наше рассуждение из раздела 7.7 о том, что по полиномиальному алгоритму можно построить полиномиальную схему, которая (для входов данной длины) будет вычислять то же самое, что и алгоритм. В нашем случае алгоритм $\mathscr{C}$ отвечает на вопрос, является ли S решением для I. Этот ответ появится на выходе схемы.

Итак, по входу I задачи A мы за полиномиальное время можем построить схему, известными входными элементами которой будут биты I, а неизвестными — биты S. На выходе такой схемы появится значение `true` тогда и только тогда, когда на неизвестные входы поданы биты, кодирующие решение S для I. Другими словами, *выполняющие наборы для схемы соответствуют решениям задачи A для входа I*, что нам и требовалось. Сведение построено.

Неразрешимые задачи

Задача из класса NP может быть решена экспоненциальным алгоритмом, который просто перебирает всех потенциальных кандидатов (напомним, что их размер ограничен полиномом от размера входа). Оказывается, что бывают задачи, которые не решаются вообще никаким алгоритмом (ни полиномиальным, ни экспоненциальным, ни ещё более долго работающим); их называют *неразрешимыми* (unsolvable).

Например, таков арифметический вариант задачи выполнимости. В нём по заданному алгебраическому уравнению типа

$$x^3yz + 2y^4z^2 - 7xy^5z = 6$$

(в левой части стоит многочлен от нескольких переменных с целыми коэффициентами) нужно определить, есть ли у него целочисленные решения.

Неразрешимость этой конкретной задачи была доказана в 1970 году (в результате работ Джулии Робинсон, Мартина Дэвиса, Хилари Патнэма и Юрия Матиясевича), но впервые неразрешимая задача была указана Аланом Тьюрингом в 1936 году; он был тогда студентом математического фа-

культета в Кембридже (Англия). В те времена ещё не было ни компьютеров, ни языков программирования (скорее наоборот, компьютеры и языки программирования появились благодаря Тьюрингу). На современном языке эту задачу можно описать так.

Допустим, нам дана программа на некотором языке программирования, а также вход этой программы. Мы хотим понять, остановится ли программа, начав работать на этом входе. Было бы очень здорово иметь процедуру TERMINATES(p, x), которая по программе p и файлу с данными x отвечала бы на вопрос, остановится ли программа p на данных x.

Но как можно было бы реализовать такую процедуру? Если вы не встречались с этой задачей раньше, попробуйте прикинуть, как можно было бы определить, зациклится ли данная программа на данном входе или нет.

На самом деле *такую процедуру написать нельзя*. Докажем это от противного. Пусть такая процедура TERMINATES(p, x) есть. Напишем тогда такую процедуру с одним параметром:

```
процедура PARADOX(z)
повторять пока TERMINATES(z, z) истинно
```

Она останавливается на входе z тогда и только тогда, когда процедура z *не останавливается*, если ей на вход дать её собственный код (z).

Наверно, вы уже видите подвох. Давайте создадим файл `Paradox` с кодом этой процедуры и выполним вызов PARADOX(`Paradox`). Остановится он или нет? Оба ответа на этот вопрос — положительный и отрицательный — приводят к противоречию. Противоречие произошло из-за того, что мы предположили существование процедуры, проверяющей остановку. Так что такой процедуры нет: как говорят, проблема останова (halting problem) неразрешима.

Из этого можно сделать вывод, что в программировании можно автоматизировать далеко не всё. Мы видели, что нельзя написать процедуру, определяющую, зацикливается ли программа на данном входе. Может быть, можно хотя бы проверить, не случится ли переполнение буфера во время исполнения программы? Тоже нет (к этой задаче сводится проблема останова — понятно, как?) Программирование всегда будет требовать дисциплины, изобретательности и остроумия.

Упражнения

8.1. *Задачи поиска и задачи оптимизации.* Для задачи коммивояжёра варианты с поиском и с оптимизацией выглядят так:

Задача поиска:
Вход: матрица расстояний, бюджет b.
Выход: цикл веса не более b, проходящий по всем вершинам графа, если такой цикл есть.

Оптимизационная задача:
Вход: матрица расстояний.
Выход: цикл минимального веса, проходящий по всем вершинам графа, если такой цикл есть.

Покажите, что если поисковую задачу коммивояжёра можно решить за полиномиальное время, то и оптимизационный вариант можно решить за полиномиальное время.

8.2. *Задачи поиска и задачи разрешения.* Пусть имеется программа, которая за полиномиальное время отвечает на вопрос, содержит ли входной граф гамильтонов цикл. Как с её помощью найти за полиномиальное время сам цикл (если он есть)?

8.3. Задача экономной выполнимости (stingy SAT) заключается в следующем: по данной формуле в КНФ и числу k требуется найти выполняющий набор, в котором не более k переменным присвоено значение `true`, если такой набор есть. Докажите, что эта задача NP-полна.

8.4. Назовём «задачей о 3-клике» частный случай задачи о клике для графов, в которых степень каждой вершины не превосходит 3.

(a) Докажите, что задача о 3-клике принадлежит классу NP.

(b) Приведём доказательство NP-полноты этой задачи. Что в нём неверно?

«Для доказательства NP-полноты 3-клики достаточно свести её к задаче о клике (ведь мы уже знаем, что задача о клике NP-полна). Поскольку задача о клике — более общая задача, сведение никак не будет менять ни данный граф G, степень каждой вершины которого не превосходит трёх, ни параметр g. Более того, решение для полученной задачи о клике, очевидно, является решением и для исходной задачи о 3-клике. Таким образом, сведение построено, а значит, задача о 3-клике является NP-полной».

(c) Задача о вершинном покрытии остаётся NP-трудной, если ограничить её графами, в которых степень каждой вершины не превосходит 3. Назовём эту задачу ВП-3. Где ошибка в следующем доказательстве NP-полноты задачи о 3-клике?

«Построим сведение задачи ВП-3 к задаче о 3-клике. Пусть дан граф $G = (V, E)$ со степенями вершин не выше 3, а также число b; подадим на вход задаче о 3-клике граф G и параметр $|V| - b$. Ясно, что множество вершин $C \subseteq V$ является вершинным покрытием в G тогда и только тогда, когда его дополнение $V - C$ является кликой в G. Значит, в G есть вершинное покрытие размера не более b тогда и только тогда, когда в нём есть клика размера хотя бы $|V| - b$. Построенное сведение доказывает NP-полноту задачи о 3-клике».

(d) Постройте алгоритм, решающий задачу о 3-клике за время $O(|V|^4)$.

8.5. Постройте сведение задачи о трёхдольном сочетании к задаче выполнимости, а также сведение задачи о гамильтоновом пути к задаче выполнимости. (Подсказка: во втором сведении можно использовать переменные x_{ij},

отвечающие за то, что i-я вершина имеет номер j в гамильтоновом пути. Используя эти переменные, нужно записать ограничения задачи.)

8.6. На странице 249 мы узнали, что задача 3-выполнимости остаётся NP-полной даже для формул, в которых каждый литерал встречается не более двух раз.

(a) Докажите, что если каждый литерал встречается не более *одного раза*, то задача может быть решена за полиномиальное время.

(b) Докажите, что задача о независимом множестве остаётся NP-полной для графов, в которых степень каждой вершины не выше 4.

8.7. Рассмотрим следующий частный случай задачи выполнимости: каждый дизъюнкт состоит ровно из трёх литералов, каждая переменная встречается не более трёх раз. Докажите, что данная задача может быть решена за полиномиальное время. (Подсказка: в двудольном графе, где слева дизъюнкты, справа переменные, а рёбра соответствуют вхождению переменной в дизъюнкт, есть совершенное паросочетание, см. упражнение 7.30.)

8.8. Входом задачи точной 4-выполнимости (exact 4SAT) является множество дизъюнктов, каждый из которых содержит ровно четыре переменные, причём все эти переменные различны. Найти, как обычно, нужно выполняющий набор. Покажите, что данная задача NP-полна.

8.9. В задаче о множестве представителей (hitting set) на вход даются семейство множеств $\{S_1, S_2, \ldots, S_n\}$ и число b. Требуется найти множество H размера не более b, пересекающее каждое S_i (то есть $S_i \cap H \neq \varnothing$ при всех i), или сообщить, что такого нет. Докажите, что эта задача NP-полна.

8.10. *Доказательство NP-полноты обобщением.* Покажите, что каждая приведённая ниже задача NP-полна, поскольку она обобщает некоторую уже известную нам NP-полную задачу.

(a) Изоморфизм подграфа (subgraph isomorphism). По двум неориентированным графам G и H выяснить, является ли G подграфом H (то есть можно ли удалить из H некоторые вершины и рёбра так, чтобы остался граф, который с точностью до названий вершин совпадает с G), и если является, указать соответствие между вершинами.

(b) Максимальный путь (longest path). Даны граф G и число g; найти в G простой путь длины не менее g.

(c) Задача максимальной выполнимости (max SAT). По данной формуле в КНФ и числу g найти набор, выполняющий хотя бы g дизъюнктов.

(d) Плотный подграф (dense subgraph). Даны граф G и числа a и b; найти a вершин в G, соединённых друг с другом хотя бы b рёбрами.

(e) Разреженный подграф (sparse subgraph). Даны граф G и числа a и b; найти a вершин в G, между которыми в G есть не более b рёбер.

(f) Покрытие множествами (set cover). Эта задача обобщает *две* известные нам NP-полные задачи.

(g) Задача о надёжной сети (reliable network). Даны две матрицы размера $n \times n$: матрица расстояний d_{ij} и матрица *требований связности* r_{ij}, а также число b. Надо найти граф $G = (\{1, 2, \ldots, n\}, E)$, в котором (1) сумма длин рёбер не превосходит b и (2) между любыми двумя вершинами i и j есть хотя бы r_{ij} путей, не пересекающихся по вершинам. (Подсказка: пусть все d_{ij} равны 1 или 2, $b = n$, а все r_{ij} равны 2. Какая NP-полная задача получается?)

8.11. Есть несколько версий задачи о гамильтоновом пути: граф может быть ориентированным или нет, искать можно путь или цикл. Сведите задачу о гамильтоновом пути в ориентированном графе к следующим двум задачам.

(a) Задача о гамильтоновом пути в неориентированном графе.

(b) Задача о гамильтоновом (s, t)-пути в неориентированном графе (ищется гамильтонов путь с заданным началом s и концом t).

8.12. Задача о k-покрывающем дереве (k-spanning tree) состоит в следующем.

Вход: неориентированный граф $G = (V, E)$.
Выход: покрывающее дерево G, в котором каждая вершина имеет степень не более k.

Покажите, что для любого $k \geqslant 2$

(a) задача о k-покрывающем дереве является задачей поиска (лежит в NP);

(b) задача о k-покрывающем дереве NP-полна. (Подсказка: начните с $k = 2$ и установите связь между этой задачей и задачей о гамильтоновом пути.)

8.13. Для каждой из перечисленных ниже задач предложите полиномиальный алгоритм или докажите NP-полноту. На вход даётся неориентированный граф $G = (V, E)$, а также

(a) множество вершин $L \subseteq V$, и надо найти покрывающее дерево, в котором все вершины из L являются листьями;

(b) множество вершин $L \subseteq V$, и надо найти покрывающее дерево, в котором листьями являются все вершины из L и только они;

(c) множество вершин $L \subseteq V$, и требуется найти покрывающее дерево, в котором множество листьев лежит в L;

(d) число k, и надо найти покрывающее дерево, у которого не более k листьев;

(e) число k, и надо найти покрывающее дерево, у которого не менее k листьев;

(f) число k, и надо найти покрывающее дерево, у которого ровно k листьев.

(Подсказка: Все задачи, кроме одной, являются непосредственными обобщениями NP-полных.)

8.14. Докажите NP-полноту следующей задачи: по данному неориентированному графу $G = (V, E)$ и числу k найти клику размера k и независимое множество размера k, если оба искомых объекта есть в графе.

8.15. Докажите, что следующая задача о максимальном общем подграфе NP-полна.

> *Вход*: два графа $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ и число b.
> *Выход*: два подмножества вершин $V_1' \subseteq V_1$, $V_2' \subseteq V_2$, после удаления которых получатся два одинаковых (изоморфных) графа с не менее чем b вершинами.

8.16. Повар-экспериментатор хочет приготовить новое блюдо, использовав максимальное количество из имеющихся ингредиентов. Некоторые из них плохо сочетаются: для каждых двух ингредиентов i и j (номера от 1 до n) в таблице есть число от 0 до 1 — штраф за их совместное использование (0 — нет проблем, 1 — максимальная несовместимость). Вот пример подобной таблицы с 5 ингредиентами:

	1	2	3	4	5
1	0,0	0,4	0,2	0,9	1,0
2	0,4	0,0	0,1	1,0	0,2
3	0,2	0,1	0,0	0,8	0,5
4	0,9	1,0	0,8	0,0	0,2
5	1,0	0,2	0,5	0,2	0,0

В данном примере ингредиенты 2 и 3 сочетаются почти идеально, а 1 и 5 — крайне плохо. Два естественных свойства таблицы: она симметрична и на диагонали стоят нули. Для любого набора ингредиентов можно вычислить штраф (сумма штрафов по всем парам; например, для $\{1, 3, 5\}$ получаем $0{,}2 + 1{,}0 + 0{,}5 = 1{,}7$). Мы хотим, чтобы штраф был небольшим. Получаем такую задачу о поваре-экспериментаторе (experimental cuisine):

> *Вход*: число n, матрица штрафов $n \times n$ и число $p \geqslant 0$.
> *Выход*: вариант с максимально возможным числом ингредиентов и суммарным штрафом не более p.

Покажите, что если эта задача решается за полиномиальное время, то за полиномиальное время можно решить и задачу 3-выполнимости.

8.17. Покажите, что для любой задачи Π из класса NP существует алгоритм, который решает Π за время $O(2^{p(n)})$, где n — размер входа, а p — некоторый полином (свой для каждой задачи).

8.18. Докажите, что если $\mathrm{P} = \mathrm{NP}$, то криптосистема RSA (раздел 1.4.2) может быть взломана за полиномиальное время.

8.19. Граф с $2n$ вершинами называется *воздушным змеем* (kite), если n его вершин образуют клику, а оставшиеся n вершин соединены в путь, начинающийся в одной из вершин клики. В задаче о воздушном змее (kite problem) по данному графу и числу g требуется проверить, есть ли в графе подграф на $2g$ вершинах, являющийся воздушным змеем. Докажите NP-полноту этой задачи.

8.20. *Доминирующим множеством* (dominating set) неориентированного графа $G=(V,E)$ называется такое подмножество его вершин $D \subseteq V$, что любая вершина графа либо лежит в D сама, либо соединена с вершиной, лежащей в D. Докажите NP-полноту данной задачи.

8.21. *Секвенирование путём гибридизации.* Пусть мы хотим восстановить последовательность аминокислот в ДНК и узнали (для данного k), какие наборы из k символов она содержит (нашли все подстроки длины k) и сколько раз встречается каждый из них.

Более формально, для строки x обозначим через $\Gamma(x)$ мультимножество всех её k-подстрок; в нём $|x|-k+1$ элементов (каждый с учётом кратности). Нам нужно по данному мультимножеству строк длины k найти строку x, для которой $\Gamma(x)$ именно такое.

(a) Сведите эту задачу к задаче о гамильтоновом пути. (Подсказка: постройте ориентированный граф, вершинами которого будут все данные подстроки, а рёбра соединяют a и b, если последние $k-1$ символов a совпадают с первыми $k-1$ символами b.)

(b) Мало того, эта задача также сводится к задаче об эйлеровом пути. (Подсказка: на этот раз данные подстроки задают рёбра графа.)

8.22. Представим последовательно выполняемые дела в виде ориентированного графа (ребро из i в j означает, что i должно предшествовать j). Иногда ограничения противоречат друг другу (если в графе есть цикл). Мы хотим удалить минимальное количество рёбер так, чтобы после этого циклов не осталось.

Подмножество рёбер $E' \subseteq E$ ориентированного графа $G=(V,E)$ называется *множеством обратной связи* (feedback arc set), если удаление этих рёбер делает граф ациклическим.

Задача: по данному ориентированному графу $G=(V,E)$ и числу b найти множество обратной связи размера не более b.

(a) Докажите, что данная задача NP-полна.

Это можно сделать, сведя к ней задачу о вершинном покрытии. Пусть дан неориентированный граф $G=(V,E)$ и мы хотим найти в нём вершинное покрытие размера не более b. Пусть $V=\{v_1,\ldots,v_n\}$. Построим ориентированный граф $G'=(V',E')$, в котором V' состоит из $2n$ вершин $w_1, w'_1, \ldots, w_n, w'_n$, а E' содержит $n+2|E|$ (ориентированных) рёбер:

- (w_i, w'_i) для всех $i=1,2,\ldots,n$;
- (w'_i, w_j) и (w'_j, w_i) для всех $(v_i, v_j) \in E$.

(b) Покажите, что если в G есть вершинное покрытие размера b, то в G' есть множество обратной связи размера b.

(c) Покажите обратное: если в G' есть множество обратной связи размера b, то в G есть вершинное покрытие размера b. (Подсказка: удобно сперва слегка переделать данное множество обратной связи размера b, не увеличивая его размер, после чего построить по нему вершинное покрытие того же размера, что и переделанное множество.)

8.23. В задаче о путях без общих вершин (node disjoint paths) на вход даётся неориентированный граф, в котором некоторые вершины $s_1, \ldots, s_k$ помечены как «истоки», а некоторые вершины $t_1, \ldots, t_k$ — как «стоки». Требуется соединить все s_i с соответствующими t_i путями, не имеющими общих вершин. Мы хотим доказать NP-полноту этой задачи; следующие пункты указывают способ это сделать (чем дальше — тем подробнее).

(a) Сведите к ней задачу 3-выполнимости.

(b) Для формулы в 3-КНФ с n переменными и m дизъюнктами используйте $k = m + n$ пар исток-сток. Заведите пару исток-сток (s_x, t_x) для каждой переменной x и пару исток-сток (s_c, t_c) для каждого дизъюнкта c.

(c) Для каждого дизъюнкта введите 6 промежуточных вершин, соответствующих литералам дизъюнкта и их отрицаниям.

(d) Если путь из s_c в t_c проходит через промежуточную вершину, соответствующую некоторому литералу, то никакой другой путь уже не может проходить через эту вершину. Как использовать это, чтобы обеспечить согласованность выбора истинных литералов в разных дизъюнктах?

Глава 9

Решение NP-полных задач

Посмотрим на ситуацию с точки зрения практики. Возникла какая-то задача про графы, и нужно написать программу, её решающую. Чем может помочь наша теория? Если повезёт, задача может оказаться вариантом какой-то из разобранных или сводиться к ним, или решаться каким-то известным приёмом (динамическое или линейное программирование, минимальные покрывающие деревья, потоки) — хотя не всегда легко это сразу заметить. Такое везение, увы, бывает редко — наугад взятая задача поиска скорее окажется NP-полной, и что тогда?

Для начала можно доказать, что ваша задача NP-полна. Возможно, какая-то из стандартных задач является её частным случаем (см. обсуждение на с. 253 и упражнение 8.10), или можно свести к ней одну из NP-полных задач (скажем, выполнимость или УНЕ). Но что это даёт? Конечно, поупражняться всегда полезно, к тому же приятно осознавать, что дело не в недостатке умения, а в сложности задачи. Но можно ли сделать что-то ещё?

Иногда можно, несмотря на доказанную NP-полноту задачи — не исключено, что на практике попадаются несложные случаи. Скажем, NP-полнота задачи выполнимости не страшна, если нас интересуют (как в логическом программировании) только хорновские формулы (см. раздел 5.3). Другой пример: если реально интересующие нас графы представляют собой деревья, то трудные для произвольных графов задачи (например, задача о независимом множестве) могут быть решены быстро при помощи динамического программирования (раздел 6.7).

К сожалению, такой подход работает не всегда — задача может остаться NP-полной и после ограничения. Скажем, задача выполнимости остаётся полной для дизъюнктов с тремя литералами, а задача о независимом множестве (и многие другие) — для планарных графов (графов, которые можно нарисовать на плоскости без пересечений). Да и понять, какие частные случаи нужны на практике, не просто. Что ж, тогда можно попытаться хотя бы оптимизировать перебор (см. раздел 9.1 о переборе с возвратом и методе ветвей и границ) — вдруг этого хватит для практически важных случаев.

С другой стороны, можно смириться с тем, что оптимального решения мы не найдём, и удовлетвориться решением, которое не *сильно хуже* оптимального. Например, в разделе 5.4 мы рассмотрели жадный алгоритм, который гарантирует, что найденное семейство покрывающих множеств отличается от оптимального не более чем в $\log n$ раз. Алгоритмы, дающие подобные гарантии, называют *приближёнными* (approximation algorithms), и они известны для многих NP-полных оптимизационных задач (см. раздел 9.2). Часто они

очень изящны (хотя и не просты). С другой стороны, и тут есть некоторый барьер: для некоторых NP-полных задач доказано, что отыскание приближённых решений с точностью лучше некоторой границы означало бы P = NP.

На худой конец бывают и «эвристические» алгоритмы, которые в принципе ничего не гарантируют (ни времени работы, ни качества приближения), но могут реально работать, если их автор достаточно изобретателен, имеет опыт работы в данной области, долго подбирал параметры и придумывал разные улучшения, или воспользовался идеей из физики или биологии. Некоторые примеры такого рода рассмотрены в разделе 9.3.

9.1. Оптимизация перебора

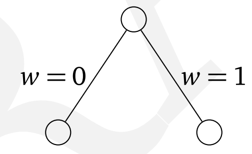
9.1.1. Перебор с возвратом

Иногда можно понять, что выбранный путь — тупиковый. Например, если мы хотим выполнить формулу, где есть дизъюнкт $(x_1 \vee x_2)$, то все наборы значений переменных с $x_1 = x_2 = 0$ (четверть всех) можно не рассматривать. Но как это делать систематически?

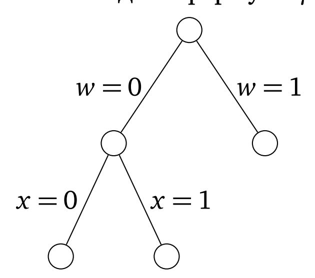
Пусть дана формула $\varphi(w, x, y, z)$ из таких дизъюнктов:

$$(w \vee x \vee y \vee z), \quad (w \vee \bar{x}), \quad (x \vee \bar{y}), \quad (y \vee \bar{z}), \quad (z \vee \bar{w}), \quad (\bar{w} \vee \bar{z}).$$

Будем подбирать значения переменных по очереди, начав с w:



Оба варианта $w = 0$ и $w = 1$ допустимы (ни один из дизъюнктов не становится заведомо ложным), так что надо оставить обе ветви поиска. Начать можно с любой, выбрав в ней следующую переменную, например, так:



На этот раз нам повезло: при $w = 0$, $x = 1$ дизъюнкт $(w \vee \bar{x})$ становится ложным, и эту ветвь поиска можно закрыть, признав тупиком. Поиск продолжается в одной из двух оставшихся активных вершин.

Таким образом, перебор с возвратом исследует возможные наборы, наращивая дерево только в тех вершинах, где выполняющий набор всё ещё возможен, и останавливается, когда такой набор найден.

Для задачи выполнимости булевой формулы в каждой вершине дерева поиска есть частичный набор значений переменных и множество дизъюнктов, остающихся при подстановке этих значений в исходную формулу. Например, если $w = 0$ и $x = 0$, то все дизъюнкты, содержащие $\overline{w}$ или $\bar{x}$, а также вхождения w и x в другие дизъюнкты могут быть удалены, так как первые всегда истинны, а вторые ложны. В итоге остаётся

$$(y \vee z), \quad (\bar{y}), \quad (y \vee \bar{z}).$$

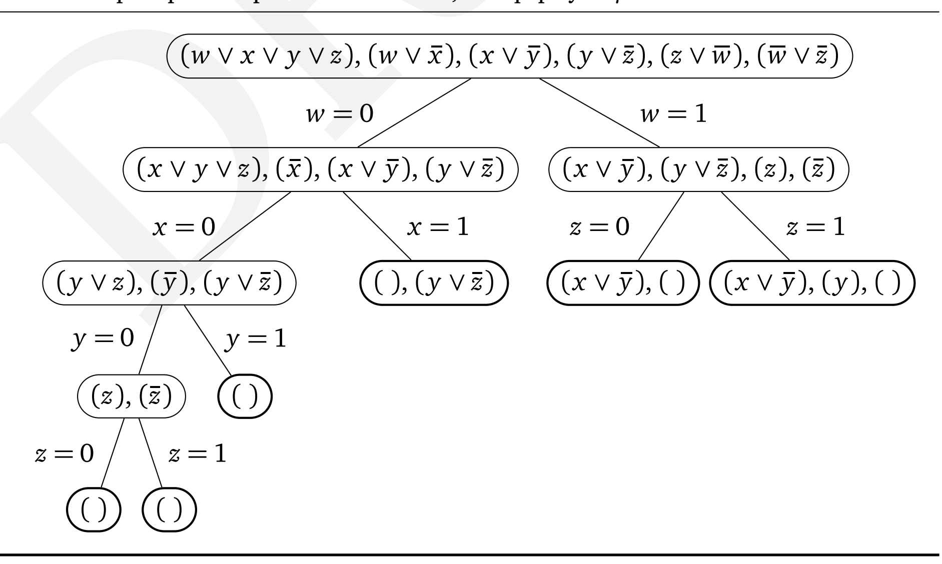
А при $w = 0$ и $x = 1$ получается

$$(\,), \quad (y \vee \bar{z}),$$

где пустой дизъюнкт $(\,)$ свидетельствует о том, что формула уже не может быть выполнена. Таким образом, подзадача для каждой вершины сама имеет вид задачи выполнимости.

На каждом шаге поиска нужно решать, какую ветвь выбрать и по какой переменной расщеплять на два случая. Хочется сократить пространство поиска, это происходит при появлении пустых дизъюнктов. Поэтому логично выбирать подзадачу с *наиболее коротким* условием-дизъюнктом и расщеплять по одной из его переменных. Если в этом дизъюнкте окажется всего один литерал, то одну из ветвей можно будет сразу отбросить. (Если это не определяет выбор однозначно, то из разных подзадач логично выбирать ту, которая ниже по дереву, поскольку она, возможно, ближе к выполняющему набору.) На рисунке 9.1 изображено итоговое дерево для рассмотренного выше примера.

Рис. 9.1. Перебор с возвратом показывает, что формула φ невыполнима.



В общем случае, алгоритм перебора с возвратом содержит *тест*, который по данной подзадаче быстро выдаёт один из трёх ответов:

1) неудача: подзадача не имеет решения;

2) успех: найдено решение подзадачи;

3) неопределённость.

В нашем примере тест даёт ответ «неудача», если текущая формула содержит пустой дизъюнкт; «успех», если формула пуста (не содержит ни одного дизъюнкта); «неизвестно» — в остальных случаях.

Общий вид алгоритма перебора с возвратами:

```
начать с некоторой задачи P_0
S ← {P_0} {множество активных подзадач}
пока S не пусто:
  выбрать подзадачу P ∈ S и удалить её из S
  разбить P на меньшие подзадачи P_1, P_2, …, P_k
  для каждой P_i:
    если тест(P_i) = успех:  вернуть найденное решение
    если тест(P_i) = неудача:  отбросить P_i
    иначе:  добавить P_i в S
сообщить, что решения нет
```

Процедура выбрать выбирает одну из подзадач, а разбить выбирает переменную для расщепления и строит две подзадачи для двух вариантов её значения. (Мы уже обсуждали способы такого выбора.)

При удачном подборе и реализации процедур тест, выбрать и разбить перебор с возвратами может быть вполне практичным: рассмотренный нами алгоритм поиска для задачи выполнимости лежит в основе многих успешно работающих на практике программ. Дополнительным подтверждением его эффективности является то, что выполнимость формул в 2-КНФ он проверяет за полиномиальное время (упражнение 9.1).

9.1.2. Метод ветвей и границ

Рассмотренный метод может быть использован не только для задач поиска (в нашем примере — задачи выполнимости), но и для задач оптимизации. Допустим, мы решаем задачу минимизации (для задачи максимизации всё аналогично).

Как и раньше, мы рассматриваем частичные решения. Нам необходимо понять, какие из них заведомо не приведут к оптимальному решению, чтобы их отбросить и сэкономить время. Отбросить подзадачу можно, если мы знаем, что на этом пути получится решение, худшее уже найденного в другой ветви. Но откуда мы это можем узнать? Для этого надо использовать *нижнюю оценку* стоимости решения.

```
начать с некоторой задачи P_0
𝒮 ← {P_0} {множество активных подзадач}
текущий_минимум ← ∞
```

```
пока 𝒮 не пусто:
  выбрать подзадачу (частичное решение) P ∈ 𝒮 и удалить её из 𝒮
  разбить P на меньшие подзадачи P_1, P_2, ..., P_k
  для каждой P_i:
    если P_i является полным решением:
      обновить текущий_минимум с учётом P_i
    иначе если нижняя_оценка(P_i) < текущий_минимум:
      добавить P_i в 𝒮
вернуть текущий_минимум
```

В качестве примера рассмотрим задачу коммивояжёра на графе $G=(V,E)$ с положительными весами d_e на рёбрах. Частичным решением является простой путь $a \rightsquigarrow b$, проходящий через множество вершин $S \subseteq V$, содержащее a и b. Частичное решение будем условно обозначать $[a, S, b]$ (на самом деле a в алгоритме не меняется). Подзадача при этом состоит в том, чтобы достроить его до полного цикла, то есть найти кратчайший путь $b \rightsquigarrow a$, проходящий по всем вершинам $V-S$. В качестве исходной задачи возьмём $[a, \{a\}, a]$ для произвольного $a \in V$.

На каждом шаге метода ветвей и границ к текущему частичному решению $[a, S, b]$ добавляется ребро (b, x) для некоторого $x \in V - S$. Количество способов выбрать x равно $|V-S|$, и каждый из них ведёт к подзадаче $[a, S \cup \{x\}, x]$.

Как же вычислить нижнюю границу стоимости для ещё не найденной части цикла в подзадаче $[a, S, b]$? Есть много способов (в том числе и сложных), мы рассмотрим один из самых простых. Остаток цикла представляет собой путь из b в a по вершинам из $V-S$, так что его длина не меньше суммы трёх слагаемых:

1) самого лёгкого ребра из a в $V-S$;

2) самого лёгкого ребра из b в $V-S$;

3) минимального покрывающего дерева графа на вершинах $V-S$.

(Понятно ли, почему?) Эту нижнюю оценку можно быстро вычислить (минимальное покрывающее дерево мы искать умеем).

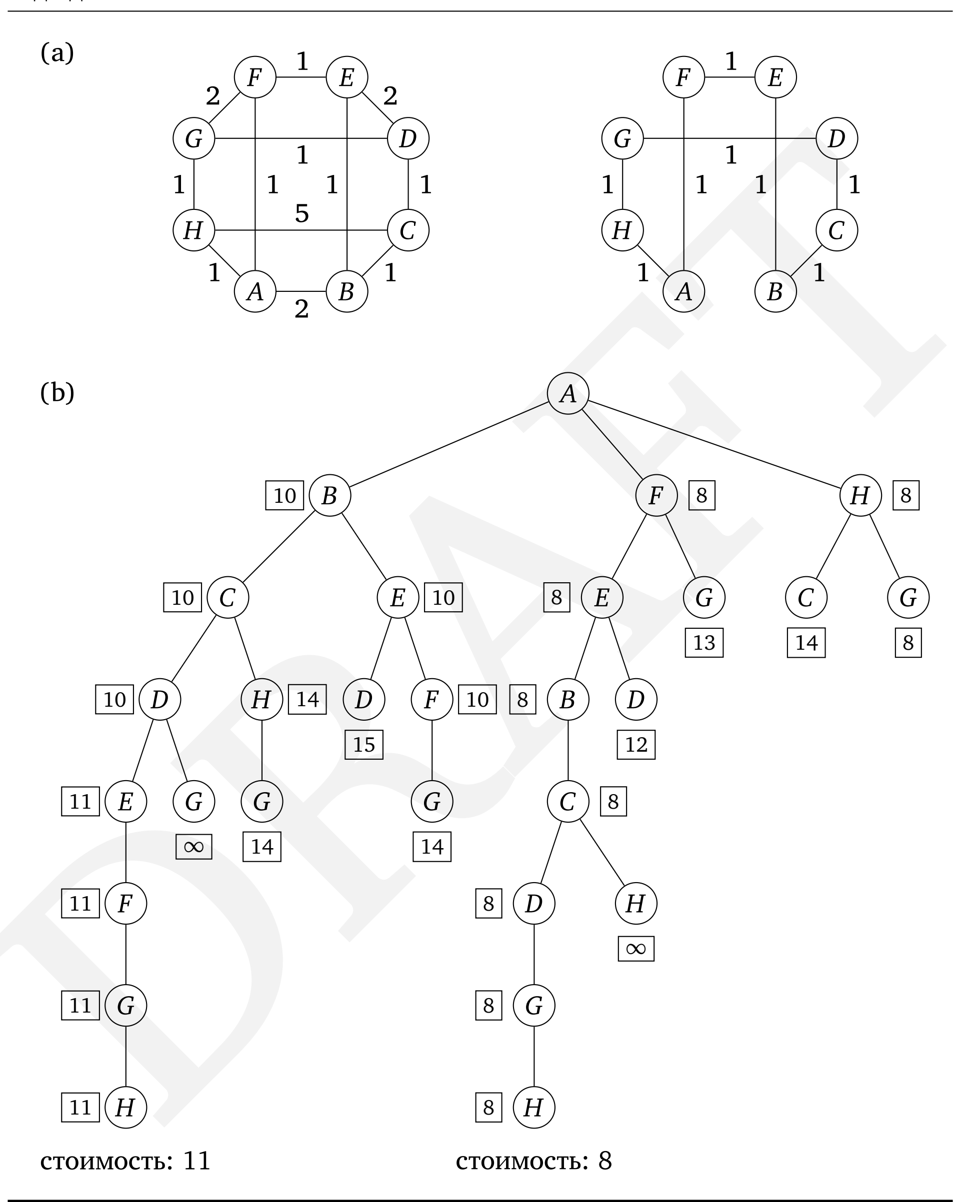
Например, в графе на рис. 9.2 удаётся найти оптимальный цикл, рассмотрев всего 28 частичных решений вместо всех возможных $7! = 5040$ полных решений. Они соответствуют вершинам дерева (путь от корня к вершине).

9.2. Приближённые алгоритмы

В задаче оптимизации для данного входа I мы ищем максимум некоторой величины (например, независимое множество максимального размера) или её минимум (как в задаче коммивояжёра). Через $\mathrm{opt}(I)$ обозначим это оптимальное значение для данного входа I. Для простоты *будем считать, что* $\mathrm{opt}(I)$ *является положительным целым числом* (обычно это не сильно меняет дело).

В разделе 5.4 мы рассмотрели (знаменитый жадный) алгоритм для задачи о покрытии множествами. Для любого множества I размера n этот алго-

Рис. 9.2. (a) Граф и оптимальный цикл коммивояжёра в нём. (b) Обход дерева поиска слева направо. Рядом с вершинами указана нижняя оценка для соответствующей подзадачи.



ритм быстро строит покрытие мощности не более чем $\mathrm{opt}(I)\log n$. Множитель $\log n$ называется ошибкой приближения алгоритма.

Рассмотрим задачу минимизации в общем случае. Пусть $\mathscr{A}$ — алгоритм, который по входу I даёт решение со значением $\mathscr{A}(I)$. *Ошибка приближения*

(approximation ratio) алгоритма $\mathscr{A}$ определяется как

$$\alpha_{\mathscr{A}} = \max_{I} \frac{\mathscr{A}(I)}{\mathrm{opt}(I)}.$$

Алгоритм $\mathscr{A}$ в таком случае называют $\alpha_{\mathscr{A}}$-приближённым.

Другими словами, $\alpha_{\mathscr{A}}$ показывает, во сколько раз результат алгоритма $\mathscr{A}$ превышает оптимальный (в худшем случае). Похожим образом определяется ошибка приближения для задач максимизации, только мы берём обратную величину (так что $\alpha_{\mathscr{A}} > 1$).

Решая NP-полную задачу оптимизации, мы радуемся, если нашли приближённый алгоритм с небольшим $\alpha_{\mathscr{A}}$: это гарантирует, что найденные им решения будут если и не оптимальными, то не сильно хуже оптимальных (которые так и останутся неизвестными). Как можно получить такие гарантии? Разберём простой пример.

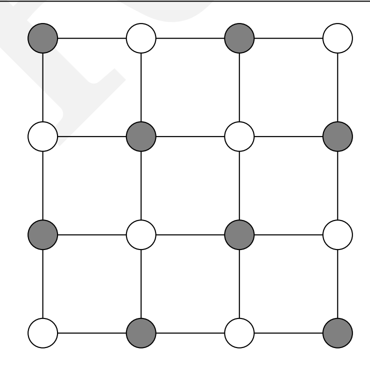
9.2.1. Вершинное покрытие

Мы уже знаем, что задача о вершинном покрытии является NP-трудной.

Вершинное покрытие
Вход: неориентированный граф $G = (V, E)$.
Выход: подмножество вершин $S \subseteq V$, «затрагивающее» все рёбра графа.
Цель: минимизировать $|S|$.

Пример входа для этой задачи и оптимальное решение для этого входа приведены на рис. 9.3.

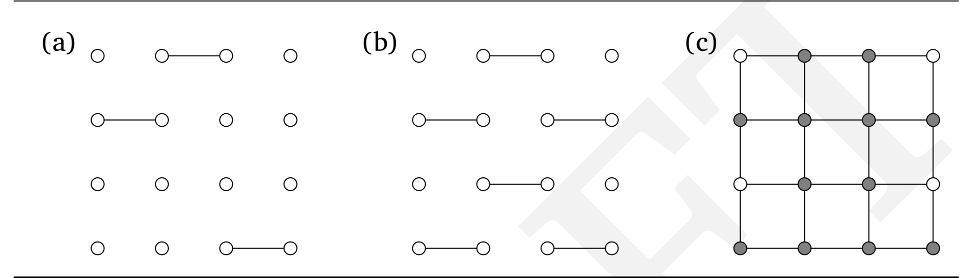
Рис. 9.3. Оптимальное вершинное покрытие графа (выделено серым цветом) имеет размер 8.



Задача о вершинном покрытии является частным случаем задачи о покрытии множествами (см. главу 5). Поэтому следующий жадный алгоритм является $O(\log n)$-приближённым: на каждом шаге выбираем вершину максимальной степени, добавляем её в вершинное покрытие и удаляем из графа. Множитель $\log n$ действительно достигается для некоторых графов.

Более точный алгоритм для этой задачи использует *паросочетания*, то есть семейства рёбер, не имеющих общих вершин (рис. 9.4). Паросочетание в графе называется *максимальным по включению*, если к нему нельзя добавить ни одного ребра. Такие паросочетания пригодятся для нахождения хорошего вершинного покрытия, и найти их легко (добавляем рёбра, не смежные с уже выбранными, пока это возможно).

Рис. 9.4. (a) Паросочетание, (b) паросочетание, достроенное до максимального, (c) полученное вершинное покрытие.



Как связаны паросочетания и вершинные покрытия? Вот простое, но важное наблюдение: мощность любого вершинного покрытия не меньше количества рёбер в паросочетании, то есть *размер паросочетания в графе I является нижней оценкой на* $\text{opt}(I)$. Действительно, каждое ребро паросочетания должно быть представлено в вершинном покрытии хотя бы одним из своих концов. (Нижние оценки важны при разработке приближенных алгоритмов, так как мы должны сравнивать качество полученного решения с оптимумом, вычисление которого является NP-полной задачей.)

Осталось сделать ещё одно наблюдение: если включить в S все концы рёбер максимального по включению паросочетания M, получится вершинное покрытие. (В самом деле, если концы какого-то ребра $e \in E$ не попали в S, то ребро e можно добавить к паросочетанию M и оно не максимально по включению.) И это покрытие с $2|M|$ вершинами не более чем вдвое уступает оптимальному (в котором не меньше $|M|$ вершин, как мы видели). Получаем такой 2-оптимальный алгоритм:

найти максимальное по включению паросочетание $M \subseteq E$
вернуть $S = \{$все концы рёбер из $M\}$

Повторим идею алгоритма: оптимальное вершинное покрытие найти сложно, но легко построить другой объект — максимальное по включению паросочетание — с двумя ключевыми свойствами:

- его размер является нижней оценкой на размер вершинного покрытия;
- на его основе можно построить вершинное покрытие, которое по размеру может оказаться больше оптимального, но разве что вдвое.

Для этого алгоритма $\alpha_{\mathscr{A}}$ в точности равно 2: несложно привести примеры, когда найденное им решение вдвое хуже оптимума.

9.2.2. Кластеризация

Рассмотрим теперь задачу кластеризации, в которой требуется разбить некоторые данные (скажем, текстовые документы, изображения или звуковые файлы) на группы схожих между собой. Для измерения «схожести» часто определяют *расстояние* между двумя элементами данных. Иногда данные с самого начала представляют собой точки в многомерном пространстве, и можно использовать обычное евклидово расстояние между ними. В других случаях расстояние определяется с помощью какого-нибудь «теста на похожесть», которому подвергаются данные. Так или иначе, считаем, что расстояние задано и что оно обладает стандартными свойствами метрики:

1. $d(x, y) \geqslant 0$ для любых x, y.
2. $d(x, y) = 0$ тогда и только тогда, когда $x = y$.
3. $d(x, y) = d(y, x)$ для любых x, y.
4. (Неравенство треугольника) $d(x, y) \leqslant d(x, z) + d(z, y)$ для любых x, y, z.

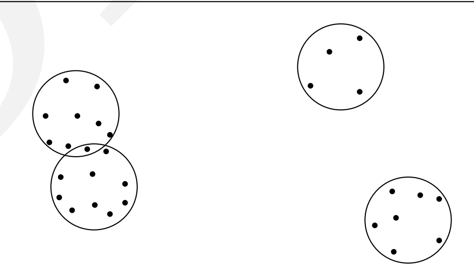
Наша задача (k-кластеризация) состоит в разбиении данных на k групп с малым диаметром.

k-кластеризация
Вход: точки $X = \{x_1, \ldots, x_n\}$ с сопутствующей метрикой $d(\cdot, \cdot)$; натуральное число k.
Выход: разбиение точек на k кластеров $C_1, \ldots, C_k$.
Цель: минимизировать диаметр кластеров, то есть

$$\max_j \max_{a,b \in C_j} d(a, b).$$

На рис. 9.5 изображён пример такого рода: набор точек (с обычным евклидовым расстоянием) делится на четыре группы.

Рис. 9.5. Лучшее разбиение точек на 4 кластера. (Точку в пересечении можно отнести к любому из двух кластеров.)

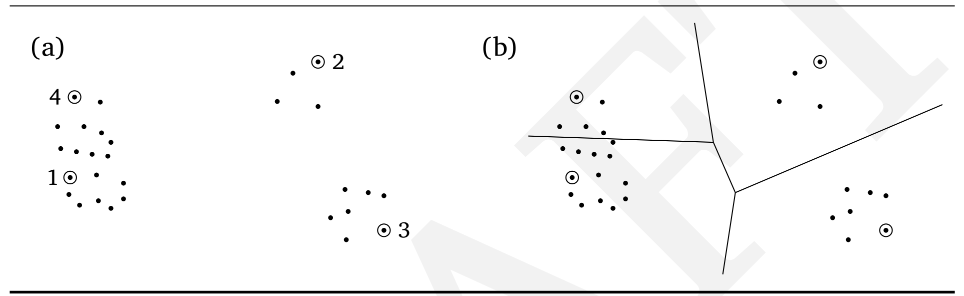


Данная задача является NP-трудной, но для неё есть простой приближённый алгоритм. Идея заключается в том, чтобы назначить некоторые k точек *центрами кластеров*, а для каждой из оставшихся точек найти ближайший центр и приписать точку к нему. Центры выбираются один за другим по та-

кому простому принципу: новым центром считаем точку, максимально удалённую от уже выбранных центров (рис. 9.6).

```
назначить произвольную точку μ₁ ∈ X центром первого кластера
для i от 2 до k:
  μᵢ ← точка из X, наиболее удалённая от μ₁, ..., μᵢ₋₁,
    то есть такая x, для которой min_{j<i} d(x, μⱼ) максимально
создать k кластеров: Cᵢ = {все x ∈ X, для которых ближайшим
центром является μᵢ}
```

Рис. 9.6. (a) Четыре центра, полученные выбором наиболее удалённых точек. (b) Полученное разбиение.



Получаем разбиение, и оказывается, что оно будет не более чем в два раза хуже оптимального. Докажем это. Пусть $x \in X$ — одна из наиболее удалённых точек от $\mu_1, \ldots, \mu_k$ (иными словами, она могла бы стать следующим центром, если бы мы разбивали на $k+1$ кластеров). Обозначим через r расстояние от x до ближайшего к ней центра. Тогда любая точка из X удалена от ближайшего к ней центра не более чем на r. Из неравенства треугольника следует, что диаметр каждого кластера не превосходит $2r$.

Всё это так, но как связано r с неизвестным нам оптимумом? А вот как: все $k+1$ точек $\mu_1, \mu_2, \ldots, \mu_k, x$ попарно удалены друг от друга хотя бы на r. (Почему?) При *любом* разбиении на k кластеров две из них попадут в один кластер, и диаметр его будет не меньше r.

Предложенный алгоритм похож на приближённый алгоритм предыдущего раздела для задачи о вершинном покрытии. Вместо максимального по включению паросочетания здесь используется другая простая структура — множество из k точек, в котором (1) каждая точка из X находится на расстоянии не более r от одного из центров, и в то же время (2) сами центры находятся друг от друга на расстоянии не меньше r. Эта структура используется и для построения кластеризации, и для нижней оценки размера оптимальных кластеров.

Лучшего алгоритма для этой задачи не известно.

9.2.3. Задача коммивояжёра

В предыдущем разделе неравенство треугольника помогло построить приближённый алгоритм для задачи k-кластеризации. Оно оказывается полез-

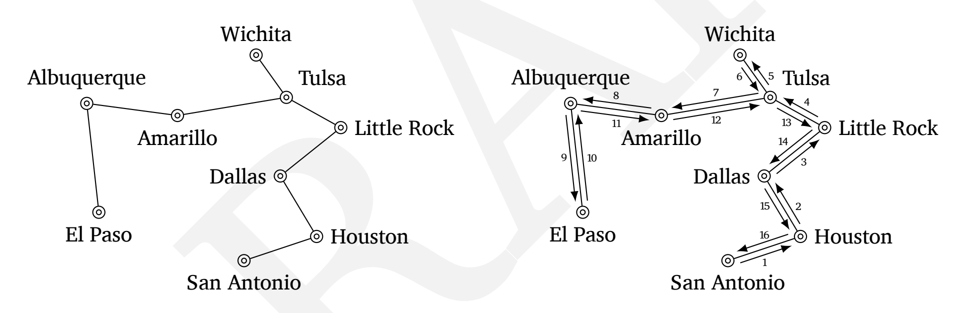
ным и для задачи коммивояжёра: если расстояния между городами обладают свойствами метрики, то существует 1,5-приближенный полиномиальный алгоритм. Мы сейчас докажем более слабое утверждение с оценкой 2.

Как и в двух предыдущих случаях, спросим себя: какая структура могла бы оказаться полезной для нахождения приближённого пути коммивояжёра? Как мы помним, такая структура должна легко вычисляться и определять некоторое решение задачи, а также некоторую нижнюю оценку на стоимость оптимального решения. Подходящую структуру легко вспомнить: это *минимальное покрывающее дерево* (МПД).

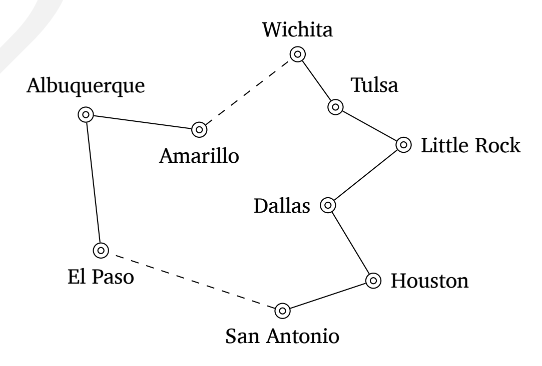
В самом деле, выкинув произвольное ребро из оптимального цикла коммивояжёра, мы получим путь, проходящий по всем вершинам, который можно рассматривать как покрывающее дерево. Значит,

$$\text{оптимум коммивояжёра} \geqslant \text{длина полученного пути} \geqslant \text{оптимум МПД}.$$

Оценка, таким образом, есть, и строить минимальное покрывающее дерево легко. Осталось научиться переделывать его в цикл коммивояжёра. Для начала просто обойдём найденное дерево, пройдя по каждому ребру *два раза*, как на рисунке (слева минимальное покрывающее дерево, а справа — его обход; числа на рёбрах указывают порядок обхода).



Длина такого цикла не превосходит удвоенной длины дерева (а тем самым и удвоенной длины оптимального цикла), но только наш цикл проходит по некоторым вершинам более одного раза. Это легко исправить: вместо того, чтобы ехать в город, в котором мы уже побывали, поедем в ближайший по нашему циклу город, в котором мы ещё не были.



Неравенство треугольника гарантирует, что от такого спрямления пути длина цикла увеличиться не может.

Общий случай задачи коммивояжёра

А что будет, если длины рёбер графа не удовлетворяют неравенству треугольника? Оказывается, в таком случае найти приближённое решение для задачи коммивояжёра *гораздо* труднее, и вот почему.

Вспомним, что на с. 257 мы построили полиномиальное сведение, которое по любому графу G и числу $C > 0$ строит вход $I(G, C)$ для задачи коммивояжёра, и при этом:

(а) если в G есть гамильтонов цикл, то $\text{opt}(I(G, C)) = n$, где n — это число вершин графа G;

(б) если же гамильтонова цикла в G нет, то $\text{opt}(I(G, C)) \geqslant n + C$.

Поскольку разница между вариантами может быть сколько угодно велика, то даже и приближённый алгоритм для задачи коммивояжёра позволил бы определять, есть ли в графе гамильтонов цикл.

Более подробно, пусть есть приближённый алгоритм $\mathscr{A}$ для задачи коммивояжёра с ошибкой $\alpha_{\mathscr{A}}$. По входу G задачи о гамильтоновом цикле построим вход $I(G, C)$ задачи коммивояжёра, положив $C = n\alpha_{\mathscr{A}}$. Запустим теперь $\mathscr{A}$ на $I(G, C)$. Глядя на длину полученного пути коммивояжёра, можно различить случаи (а) и (б): в первом случае алгоритм даёт путь длины не более $\alpha_{\mathscr{A}} \text{ opt}(I(G, C)) = n\alpha_{\mathscr{A}}$, а во втором длина пути не меньше $\text{opt}(I(G, C)) > > n\alpha_{\mathscr{A}}$. Получаем такой алгоритм, выясняющий, есть ли в графе G гамильтонов цикл:

```
построить граф I(G, C) для C = nα_𝒜 и запустить на нём алгоритм 𝒜
если найденный путь имеет длину не более nα_𝒜:
  вернуть «в G есть гамильтонов путь»
иначе: «в G нет гамильтонова пути»
```

Как мы уже знаем (упражнение 8.2), запустив такой алгоритм полиномиальное число раз, можно найти и сам путь.

Итак, если для задачи коммивояжёра существует приближённый полиномиальный алгоритм, то для задачи о гамильтоновом пути существует точный полиномиальный алгоритм. Значит, если $\text{P} \neq \text{NP}$, то приближённого алгоритма для задачи коммивояжёра не существует.

9.2.4. Задача о рюкзаке

Наш последний пример приближённого алгоритма, который мы рассмотрим в этом разделе, обладает замечательным свойством: для любого $\varepsilon > 0$ он позволяет найти решение, уступающее максимуму не более чем на множитель $1 - \varepsilon$, причём требуемое время полиномиально зависит от размера входа и от $1/\varepsilon$.

Мы уже встречались в главе 6 с задачей о рюкзаке. Есть n предметов, известны их веса $w_1, \ldots, w_n$ и стоимости $v_1, \ldots, v_n$ (все числа целые); нужно вы-

брать предметы максимальной стоимости с общим весом не больше некоторой границы W.

Мы уже знаем, что данную задачу можно решить за время $O(nW)$ методом динамического программирования. Подобным же методом можно получить верхнюю оценку $O(nV)$, где V — суммарная стоимость всех предметов. Обе эти оценки не дают полиномиального алгоритма, потому что и W, и V могут зависеть экспоненциально от размера входа.

Рассмотрим алгоритм со временем работы $O(nV)$. Если V велико, попробуем перевести все стоимости в другой масштаб. Например, вместо

$$v_1 = 117\,586\,003, \quad v_2 = 738\,493\,291, \quad v_3 = 238\,827\,453$$

будем использовать 117, 738 и 238. Это не сильно повлияет на решение, но зато сильно ускорит работу. (Масштаб зависит от требуемой точности приближения, параметра ε.) Более подробно, рассмотрим такой алгоритм:

выкинуть все предметы тяжелее W
$v_{\max} \leftarrow \max\limits_i v_i$
вычислить новые стоимости: $\hat{v}_i \leftarrow \left\lfloor \frac{n}{\varepsilon} \cdot \frac{v_i}{v_{\max}} \right\rfloor$
запустить $O(nV)$-алгоритм для $\{\hat{v}_i\}$ вместо $\{v_i\}$
вернуть найденный набор предметов

Поскольку новые стоимости $\hat{v}_i$ не превосходят n/ε, время работы полученного алгоритма есть $O(n^3/\varepsilon)$. Поймём теперь, во сколько раз найденное решение может отличаться от оптимального.

Пусть оптимальным решением исходной задачи является набор предметов S общей стоимости K^*. Стоимость набора S с точки зрения новой задачи будет равна

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} \left\lfloor \frac{v_i n}{\varepsilon v_{\max}} \right\rfloor \geqslant \sum_{i \in S} \left(v_i \cdot \frac{n}{\varepsilon v_{\max}} - 1 \right) \geqslant K^* \frac{n}{\varepsilon v_{\max}} - n.$$

Значит, стоимость оптимального набора $\hat{S}$ для новой задачи будет не меньше этого значения. Рассмотрим теперь стоимость набора $\hat{S}$ для исходной задачи:

$$\sum_{i \in \hat{S}} v_i \geqslant \sum_{i \in \hat{S}} \hat{v}_i \cdot \frac{\varepsilon v_{\max}}{n} \geqslant \left(K^* \cdot \frac{n}{\varepsilon v_{\max}} - n \right) \cdot \frac{\varepsilon v_{\max}}{n} = K^* - \varepsilon v_{\max} \geqslant K^*(1 - \varepsilon).$$

9.2.5. Иерархия приближаемости

Пусть дана некоторая оптимизационная задача. Лучше всего найти точный полиномиальный алгоритм для её решения. Если это не получается (например, для NP-полных задач), надо искать приближённый полиномиальный алгоритм с минимально возможной ошибкой. Иногда это получается лучше, иногда хуже (мы уже видели, скажем, что для общей задачи коммивояжёра алгоритм с полиномиальной ошибкой невозможен, если $P \neq NP$). С этой точ-

ки зрения можно выделить несколько классов NP-полных оптимизационных задач:

- Задачи, которые остаются трудными для любой (полиномиальной) границы ошибки (например, общий случай задачи коммивояжёра).
- Задачи, для которых есть α-приближённые алгоритмы с некоторым постоянным $\alpha > 1$, но не всякое $\alpha > 1$ достижимо. Таковы, например, задача о вершинном покрытии, задача k-кластеризации и задача коммивояжёра в метрическом пространстве. (Соответствующие нижние оценки для α мы не рассматривали; они являются одними из самых глубоких результатов в этой области.)
- К следующему классу относятся задачи, для которых α можно сделать сколь угодно близким к 1 (например, задача о рюкзаке).
- Наконец, есть и ещё один класс, попадающий между первыми двумя. Для задач этого класса известны только алгоритмы с растущей погрешностью (скажем, с множителем $\log n$). Примером такой задачи является задача о покрытии множествами.

Напомним, что вся эта классификация имеет смысл только в предположении $\mathrm{P} \neq \mathrm{NP}$. В противном случае вся эта иерархия схлопывается до класса P, и все (даже и NP-полные) оптимизационные задачи могут быть точно решены за полиномиальное время.

В заключение отметим, что на практике приближённые алгоритмы зачастую дают гораздо лучшие решения, чем можно было бы ожидать, ориентируясь на оценки точности в худшем случае.

9.3. Эвристики локального поиска

Локальный поиск (local search) — идея, которую можно применять к любой оптимизационной задаче: будем пытаться локально менять наше текущее решение, постепенно улучшая его. Для задач минимизации это выглядит так:

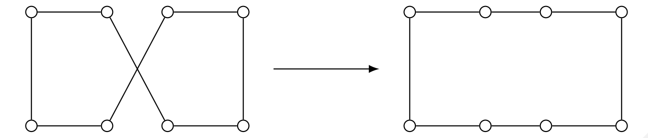
```
s ← какое-нибудь начальное решение
пока в окрестности s есть решение s′ меньшей стоимости:
  заменить s на s′
вернуть s
```

На каждом шаге текущее решение заменяется на более выгодное решение из его *окрестности* (neighborhood). Ключевой момент состоит в выборе подходящих окрестностей; для примера снова рассмотрим задачу коммивояжёра.

9.3.1. Ещё раз о задаче коммивояжёра

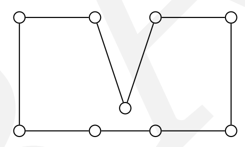
В задаче коммивояжёра с n городами пространство поиска состоит из $(n-1)!$ различных циклов. Как определить понятие окрестности? Какие циклы считать близкими? Самый естественный способ — смотреть, на сколько рёбер они отличаются. Ясно, что циклы не могут отличаться ровно в одном ребре, но разница в два ребра вполне возможна. Поэтому назовём 2-*окрестностью*

(2-change neighborhood) пути s множество всех путей, которые могут быть получены из s заменой двух рёбер на другие два ребра. Вот пример такой замены:

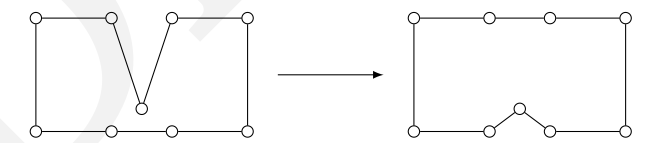


Тем самым определяется алгоритм локального поиска для задачи коммивояжёра, и вопрос теперь в том, каково его время работы и всегда ли он находит оптимальное решение.

К сожалению, ни на один из этих вопросов мы не можем убедительно ответить. Каждая отдельная итерация выполняется быстро, поскольку окрестность любого пути содержит $O(n^2)$ элементов. Непонятно, однако, сколько всего понадобится итераций, чтобы дойти до минимума. Более того, даже когда мы до него и дойдём, мы сможем утверждать лишь, что найденный путь *локально оптимален*, то есть он не хуже всех путей из его окрестности. В то же время могут быть более выгодные пути, находящиеся вдали от текущего. На рисунке ниже показан пример такого рода: изображённый путь не является оптимальным, хотя и является оптимальным в своей 2-окрестности, и 2-изменениями его не улучшить.

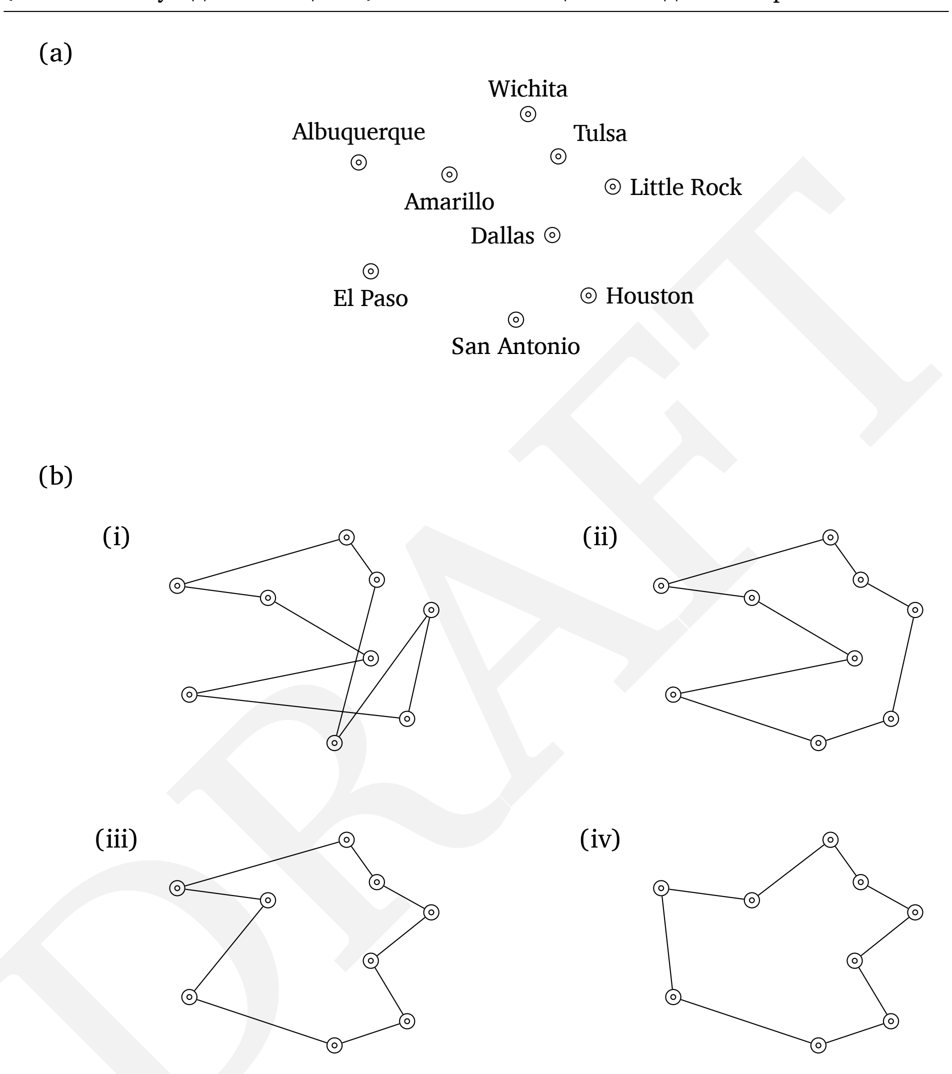


Можно попробовать увеличить окрестности, разрешив заменять три ребра. В нашем примере этого хватит: в 3-окрестности сразу найдётся оптимальный путь.



Зато теперь на каждой итерации нам нужно будет перебирать $O(n^3)$ соседей, да и проблемы это полностью не решит: для такого поиска в 3-окрестности тоже бывают локальные минимумы, не являющиеся глобальными. Можно было бы перейти, скажем, к 4-окрестностям, но тогда каждая итерация станет ещё медленнее. Как видно, за более высокое качество ответа нам приходится платить быстродействием. Чтобы алгоритм работал быстро, нужно, чтобы окружения не были слишком большими — но чем они меньше, тем вероятнее застрять в локальном минимуме, далёком от оптимума. Компромиссный вариант приходится подбирать экспериментально.

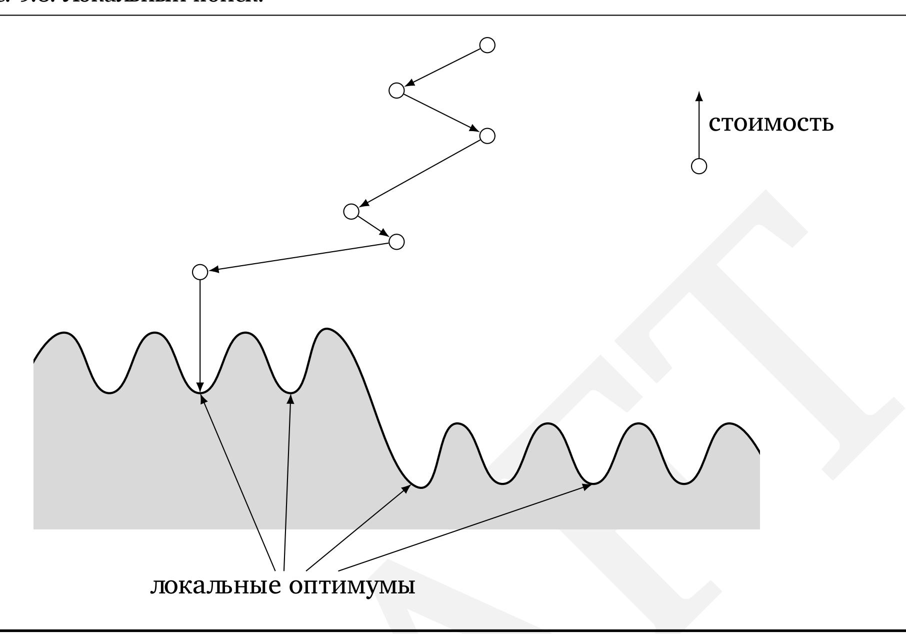
Рис. 9.7. (a) Девять городов США. (b) Поиск в 3-окрестности для задачи коммивояжёра (начатый с наугад взятого цикла). Оптимальный цикл находится за три шага.



На рисунке 9.7 показан локальный поиск для задачи коммивояжёра на ранее рассмотренном нами графе, а рисунок 9.8 изображает работу подобных алгоритмов символически. Незакрашенная область — это пространство решений, и их стоимость убывает сверху вниз. Алгоритм двигается вниз от начального решения, пока не дойдёт до локального минимума.

В общем случае, пространство решений может содержать огромное количество локальных оптимумов, сильно отличающихся от оптимального решения. Поэтому рассчитывать приходится на то, что при разумном выборе

Рис. 9.8. Локальный поиск.



структуры окрестностей большинство локальных оптимумов будут давать достаточно хорошие решения. Трудно сказать, так ли это на самом деле или нам так только кажется, но и впрямь для многих оптимизационных задач метод локального поиска оказывается одним из самых успешных на практике. В следующем разделе мы рассмотрим ещё один пример такого рода.

9.3.2. Разбиение графа

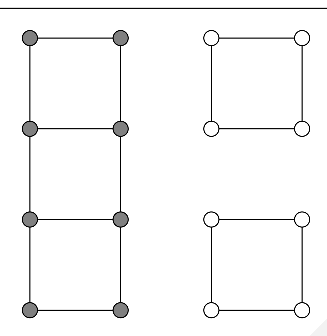
Задача разбиения графа возникает во многих приложениях — от проектирования микросхем до анализа программ и сегментации изображений. Её частным случаем является задача о сбалансированном разрезе (глава 8).

> Разбиение графа
> *Вход*: неориентированный граф $G = (V, E)$ с неотрицательными весами на рёбрах и вещественное число $\alpha \in (0, 1/2]$.
> *Выход*: разбиение вершин на две части A и B, размер каждой из которых не меньше $\alpha|V|$.
> *Цель*: минимизировать размер разреза (A, B).

На рисунке 9.9 показан пример графа с 16 вершинами, все рёбра которого имеют вес 1. Стоимость оптимального решения для данного графа равна 0.

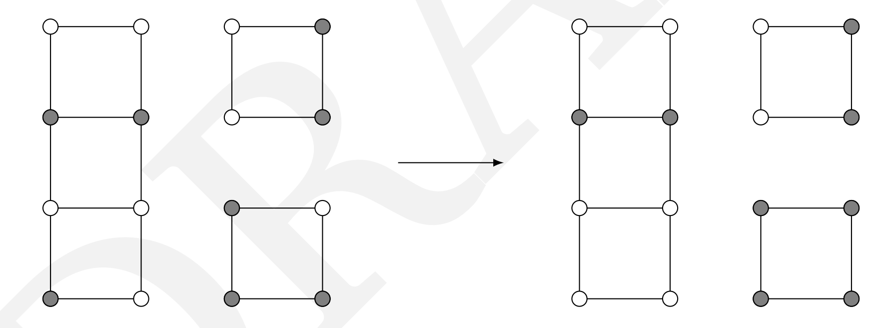
Если убрать ограничение на размер частей разбиения, то получится уже знакомая нам задача о минимальном разрезе, которая эффективно решается с помощью задачи о потоке. Однако задача о разбиении графа NP-трудна. Строя алгоритм локального поиска для этой задачи, удобно предполагать,

Рис. 9.9. Вход задачи о разбиении графа и оптимальный разрез для $\alpha = 1/2$ (разделяет белые и серые вершины).

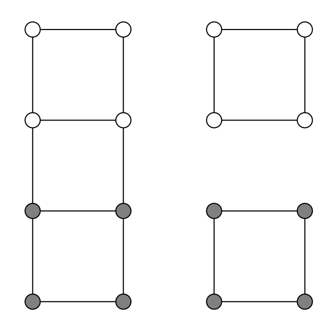


что $\alpha = 1/2$ (ищем разбиение на равные части); это не очень существенно, потому что общий случай сводится к этому частному.

Алгоритм локального поиска задаётся описанием окрестностей. Рассмотрим разрез (A, B) с $|A| = |B|$. Назовём его соседом (включим в окрестность) любой разрез, полученный обменом двух вершин между частями, то есть разрез вида $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$, где $a \in A$, $b \in B$. На рисунке показан такой обмен.



Насколько хорош такой алгоритм локального поиска? К сожалению, бывают локальные минимумы, сильно отличающиеся от глобального. Ниже показан пример такого локального минимума стоимости 2.



Что тут можно поделать? Даже разрешив менять не по одной, а по две вершины, мы не поможем делу. Вместо этого давайте рассмотрим два других общих рецепта улучшения локального поиска.

9.3.3. Способы усовершенствования локального поиска

Рандомизация и перезапуски

Рандомизацию (случайный выбор) можно использовать и при выборе начальной точки (скажем, взяв случайное разбиение вершин на части), и при выборе очередного улучшения (если есть несколько возможностей).

Благодаря этому появляется ненулевая вероятность попасть в глобальный минимум, даже если локальных минимумов много. Если эта вероятность равна p, то можно запустить алгоритм $O(1/p)$ раз и выбрать наилучший ответ (что с большой вероятностью даст глобальный минимум, см. упражнение 1.34).

На рисунке 9.10 показано пространство поиска в задаче о разбиении графа (для небольшого графа с 8 вершинами). Всего есть $C_8^4 = 70$ разбиений, но для каждого разбиения есть парное, в котором части переставлены, поэтому на рисунке показаны только 35 разбиений. Для удобства они объединены в семь групп. Есть пять локальных минимумов, из которых только один глобальный. Если начать локальный поиск в случайной точке и на каждом шаге выбирать случайный ход из наиболее выгодных, то вероятность попасть в плохой локальный минимум не более чем в четыре раза превосходит вероятность попасть в хороший. Поэтому будет достаточно небольшого количества перезапусков.

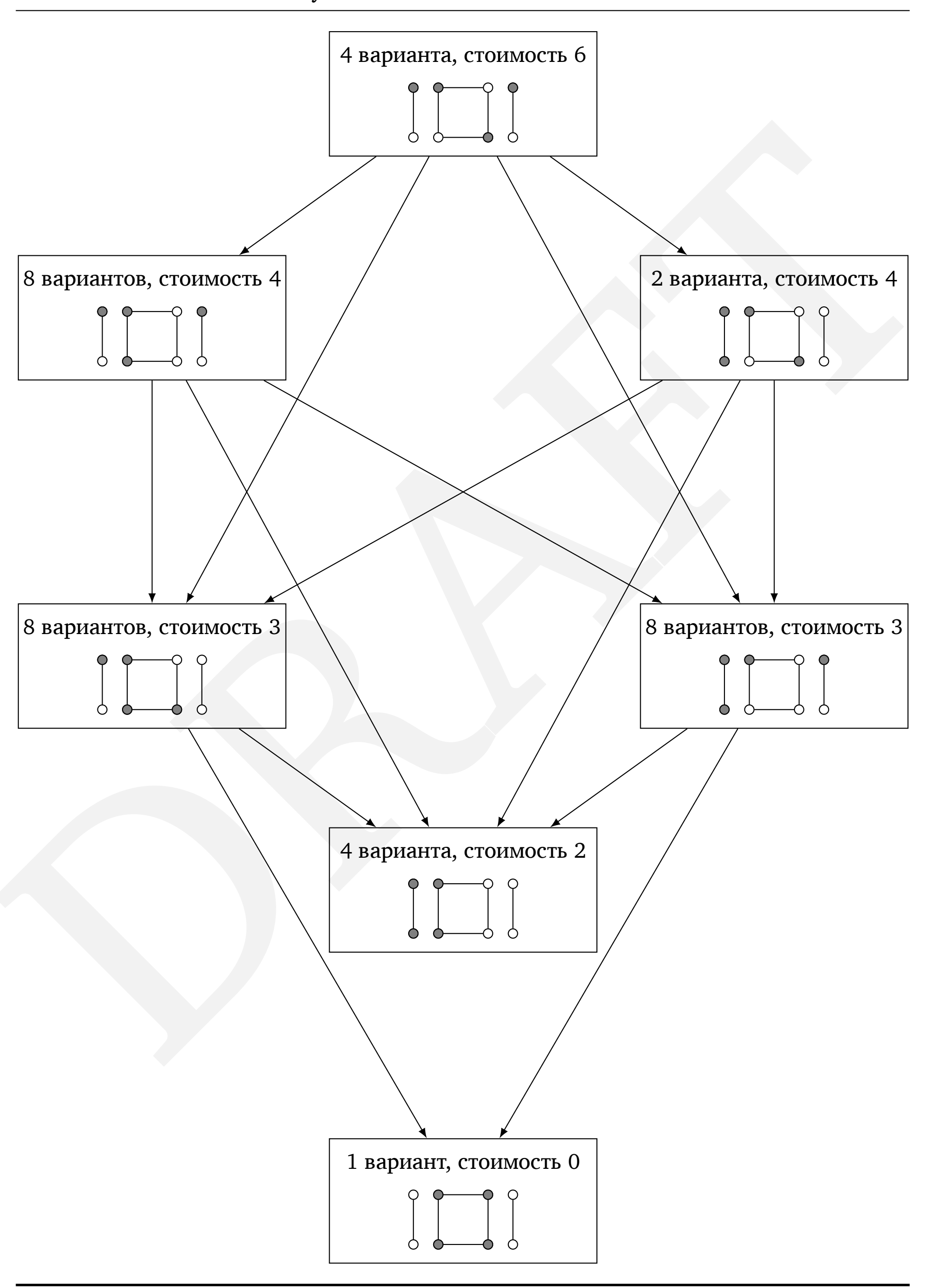
Метод имитации отжига

В примере на рис. 9.10 вероятность попасть в локальный минимум (при одном запуске алгоритма) не так уж мала. Но при увеличении размера задачи она может убывать экспоненциально, поэтому просто перезапустить алгоритм небольшое число раз уже не достаточно.

Вместо этого можно разрешать (иногда) переходить к худшим решениям: теоретически это может помочь выбраться из плохих локальных минимумов. На рис. 9.10 это могло бы помочь выйти из плохого локального минимума и перейти в хороший. Но как выбрать вероятности? *Метод имитации отжига* (simulated annealing) основан на физической аналогии и вводит неотрицательный параметр T, называемый *температурой*:

```
s ← какое-нибудь начальное решение
повторять:
  выбрать случайное решение s′ из окружения s
  Δ ← cost(s′) − cost(s)
  если Δ < 0:
    заменить s на s′
  иначе:
    заменить s на s′ с вероятностью e^{−Δ/T}
```

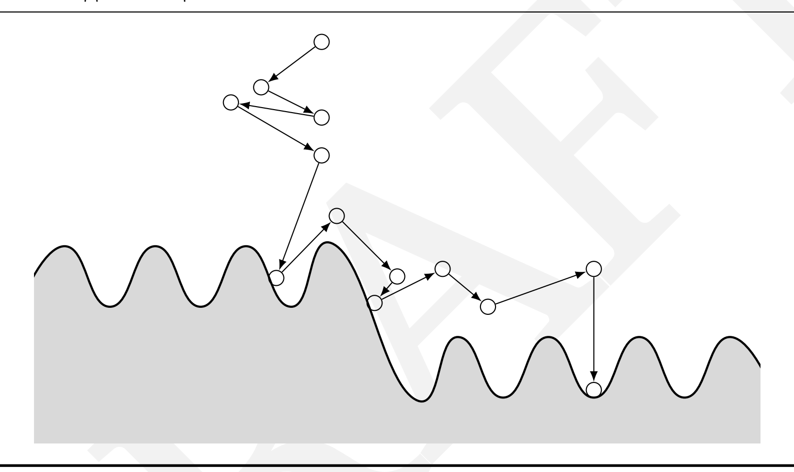
Рис. 9.10. Пространство поиска для графа на восьми вершинах. Всего есть 35 вариантов. Они разбиты на семь групп, и для каждой группы показан один из вариантов. Есть пять локальных минимумов.



Для нулевого T (считаем, что тогда $e^{-\Delta/T} = 0$) последняя строка не выполняется, и этот алгоритм превращается в рассмотренный ранее. А для больших T этот алгоритм иногда будет делать ходы в сторону ухудшения. И какое же T нам выбрать?

Хитрость тут в следующем: мы начинаем с большого значения температуры T, а потом постепенно уменьшаем её до нуля. Сначала локальный поиск будет почти что случайным блужданием, но постепенно всё больше и больше станет предпочитать ходы в сторону меньшей стоимости и лишь изредка будет делать ходы в другую сторону. В некоторой момент процесс сойдётся к локальному минимуму, и его можно будет остановить. Пример такого рода движения изображён на рис. 9.11.

Рис. 9.11. Метод имитации отжига.



Данный метод называют «имитацией отжига». Как видно из названия, он исходит из аналогии с отжигом в металлургии (нагревание с последующим медленным охлаждением). Сначала, когда вещество почти что жидкое, атомы движутся почти свободно. При понижении температуры они постепенно располагаются всё более и более регулярно и наконец выстраиваются в кристаллическую решётку (близкую к минимуму энергии).

Конечно, имитация отжига требует больше времени, чем простой локальный поиск. Более того, выбор подходящей *схемы отжига* (annealing schedule), то есть стратегии понижения температуры, является достаточно творческой задачей. Но во многих случаях это существенно улучшает найденные решения и того стоит.

Упражнения

9.1. Допустим, что в алгоритме поиска с возвратом для задачи выполнимости мы всегда выбираем подзадачу (формулу в КНФ), содержащую самый короткий дизъюнкт, и расщепляем её по переменной этого дизъюнкта. Докажите,

что время работы такого алгоритма на формулах с дизъюнктами длины 2 (задача 2-выполнимости) полиномиально.

9.2. Предложите алгоритм перебора с возвратом для задачи поиска гамильтонова пути из заданной вершины s. Укажите, что будет подзадачами, как выбирается одна из них и как она разбивается на части.

9.3. Примените метод ветвей и границ к задаче о покрытии множествами. Что будет подзадачами, как выбирается одна из них, как она разбивается на части и как вычисляется нижняя оценка?

Как вы думаете, насколько ваш алгоритм будет эффективен в типичных примерах? Почему?

9.4. Дан неориентированный граф $G = (V, E)$, в котором степень каждой вершины не больше d. Как быстро найти в нём независимое множество, размер которого составляет не менее $1/(d+1)$ от размера максимального независимого множества?

9.5. *Локальный поиск для минимального покрывающего дерева.* Рассмотрим множество всех покрывающих деревьев (не обязательно минимальных) для данного неориентированного связного графа $G = (V, E)$ с весами на рёбрах.

Из раздела 5.1 мы знаем, что добавление ребра e к покрывающему дереву T создаёт цикл. Удалив любое другое ребро $e' \neq e$ из этого цикла, получим другое покрывающее дерево T'. Будем говорить, что деревья T и T' получаются друг из друга *заменой* (e, e') и считать такие деревья *соседними*.

(a) Покажите, что любое покрывающее дерево может быть получено из любого другого покрывающего дерева последовательностью замен (переходом от соседа к соседу). Сколько таких замен может потребоваться?

(b) Пусть T' — минимальное покрывающее дерево, а T — произвольное. Покажите, что можно найти последовательность замен, переводящую T в T', для которой стоимости (cost) деревьев не возрастают: в возникающей последовательности

$$T = T_0 \to T_1 \to T_2 \to \ldots \to T_k = T'$$

неравенство $\mathrm{cost}(T_i) \geqslant \mathrm{cost}(T_{i+1})$ выполнено для всех $i < k$.

(c) Рассмотрим следующий алгоритм локального поиска, который получает на вход неориентированный граф G.

$T \leftarrow$ `какое-нибудь покрывающее дерево графа` G
`пока есть замена рёбер` (e, e')`, уменьшающая` $\mathrm{cost}(T)$:
  $T \leftarrow T + e - e'$
`вернуть` T

Докажите, что данный алгоритм всегда возвращает минимальное покрывающее дерево. Какое максимальное число итераций ему может для этого понадобиться?

9.6. На вход в задаче о дереве Штейнера (minimum Steiner tree) даётся полный неориентированный граф $G = (V, E)$ с весами d_{uv} на рёбрах и множество

терминальных вершин (terminal nodes) $V' \subseteq V$. Найти необходимо дерево минимального веса, содержащее все терминальные вершины V'. Такое дерево может содержать также и вершины не из V'.



Допустим, что веса рёбер удовлетворяют свойствам метрики (с. 275). Докажите, что тогда следующий простой алгоритм является 2-приближённым: выкинем из графа все нетерминальные вершины и найдём минимальное покрывающее дерево. (Подсказка: вспомните рассмотренный нами приближённый алгоритм для задачи коммивояжёра.)

9.7. Входом задачи о мультиразрезе является неориентированный граф $G = (V, E)$ и множество терминальных вершин $s_1, \ldots, s_k \in V$. Найти требуется минимальное по размеру множество рёбер, после удаления которых никакие две терминальные вершины не будут лежать в одной компоненте связности.

(a) Покажите, что данная задача может быть решена за полиномиальное время в частном случае, когда $k = 2$.

(b) Постройте 2-приближённый алгоритм для случая $k = 3$.

(c) Предложите алгоритм локального поиска для данной задачи.

9.8. В задаче максимальной выполнимости (maximum satisfiability) дано множество дизъюнктов и требуется найти значения переменных, при которых выполнено максимальное число дизъюнктов.

(a) Покажите, что если эту задачу можно решить (точно) за полиномиальное время, то за полиномиальное время можно решить и задачу выполнимости.

(b) Рассмотрим простой алгоритм, присваивающий каждой переменной равновероятно и независимо значение 0 или 1. Пусть входная формула содержит m дизъюнктов и k_j обозначает количество литералов в j-м дизъюнкте. Покажите, что *математическое ожидание* количества выполненных дизъюнктов равно

$$\sum_{i=1}^{m} \left(1 - \frac{1}{2^{k_j}}\right) \geqslant \frac{m}{2}.$$

Другими словами, данный алгоритм является 2-приближённым (если распространить определение на вероятностные алгоритмы, взяв среднее). Более того, если длина всех дизъюнктов равна k, то данная оценка улучшается до $1 + 1/(2^k - 1)$.

(c) Как переделать данный алгоритм в детерминированный (избавиться от случайных битов), сохранив качество приближения? (Подсказка: вместо того, чтобы выбирать значение каждой следующей переменной случай-

но, выбирайте то значение, при котором условное математическое ожидание числа выполненных дизъюнктов наибольшее.)

9.9. В задаче о максимальном разрезе (maximum cut) требуется по данному неориентированному графу $G = (V, E)$ с весами $w(e)$ на рёбрах найти такое разбиение его вершин на две части, при котором суммарный вес рёбер, соединяющих эти две части, был бы *максимален*.

Для подмножества вершин $S \subseteq V$ определим $w(S)$ как сумму $w(e)$ для всех рёбер $e = \{u, v\}$, у которых $|S \cap \{u, v\}| = 1$. Задача о максимальном разрезе состоит в максимизации $w(S)$.

Рассмотрим следующий алгоритм локального поиска, в котором соседи множества получаются изменением (добавлением или удалением) одного элемента:

```
S ← какое-нибудь множество вершин
пока есть S′ ⊆ V такое, что |(S′ − S) ∪ (S − S′)| = 1 и w(S′) > w(S):
  S ← S′
```

(a) Покажите, что данный алгоритм является 2-приближённым.

(b) Является ли данный алгоритм полиномиальным по времени?

9.10. Назовём алгоритм локального поиска *точным*, если он всегда даёт оптимальное решение. Например, алгоритм для задачи о минимальном покрывающем дереве из упражнение 9.5 является точным. Симплекс-метод, решающий задачу линейного программирования, тоже можно считать точным алгоритмом локального поиска.

(a) Покажите, что алгоритм поиска в 2-окрестности для задачи коммивояжёра не является точным.

(b) Докажите то же самое для $\lceil n/2 \rceil$-окрестности, где n — количество городов.

(c) Покажите, что алгоритм поиска в $(n-1)$-окрестности является точным.

(d) Для оптимизационной задачи A назовём A-улучшением следующую задачу поиска: по данному входу x задачи A и данному решению s для входа x найти другое решение для x с лучшей стоимостью (или же сообщить, что s оптимально и поэтому лучшего решения нет). Например, задача улучшения цикла коммивояжёра выглядит так: дана матрица расстояний и гамильтонов цикл, и нужно найти лучший цикл. Оказывается, что эта задача NP-полна, как и задача улучшения покрытия множествами. Как это доказать?

(e) Скажем, что алгоритм локального поиска имеет *полиномиальную итерацию*, если каждый шаг улучшения в этом алгоритме занимает полиномиальное время. (Например, при естественной реализации алгоритма локального поиска в $(n-1)$-окрестности для задачи коммивояжёра такого не будет.) Докажите, что если $\mathrm{P} \neq \mathrm{NP}$, то для задачи коммивояжёра и задачи о покрытии множествами нет алгоритмов локального поиска с полиномиальной итерацией.

Глава 10

Квантовые алгоритмы

Мы начали эту книгу со старинных алгоритмов для сложения и умножения чисел и с классической трудной задачи (разложение на множители, factoring). В этой главе — последней главе книги — всё будет наоборот: мы расскажем о недавно придуманном алгоритме, и это будет быстрый алгоритм разложения на множители!

Конечно, тут есть тонкость: это будет алгоритм для *квантового компьютера*.

Поведение элементарных частиц и других микроскопических объектов описывается красивой, но загадочной теорией — квантовой механикой. В 1990-е годы было обнаружено, что законы квантовой механики открывают возможность для быстрых вычислений — в том числе экспоненциально более быстрых. Как мы увидим, квантовые компьютеры (если они будут действовать так, как сейчас предполагается) смогут разлагать числа на множители за полиномиальное время. Пока такие компьютеры ещё не сделаны — но когда и если они появятся, многое изменится, и не только в лучшую сторону. Например, современные системы защиты информации при передаче по интернету и в банках (большей частью используется RSA) утратят надёжность.

10.1. Кубиты, суперпозиция, измерения

В этом разделе мы кратко опишем основные принципы квантовой механики, на которых основано действие квантовых компьютеров.[1]

В обычных компьютерах два значения бита (нуль или единица) закодированы как низкое и высокое напряжение в проводниках схемы. Можно было бы кодировать их иначе — например, атом водорода в основном состоянии (с малой энергией) может кодировать нуль, а в возбуждённом состоянии (с большой энергией) — единицу.

Используя традиционные обозначения квантовой механики, мы записываем основное состояние (ground state) как $|0\rangle$, а возбуждённое состояние (excited state) как $|1\rangle$. С точки зрения классической физики, атом может находиться либо в одном состоянии, либо в другом. В квантовой

[1]Рассказывают, что знаменитый физик Ричард Фейнман как-то сказал: «Думаю, что можно смело утверждать, что квантовую механику не понимает никто». Так что вряд ли чтение этого раздела позволит вам в ней разобраться как следует — но в конце книги мы указываем несколько источников для дальнейшего чтения. [Мы благодарим М. Н. Вялого за помощь в переводе этой главы. — *Прим. перев.*]

механике — как это ни странно на первый взгляд — возможны и *суперпозиции* (superpositions) этих состояний: формальные линейные комбинации $\alpha_0|0\rangle + \alpha_1|1\rangle$, где α_0 и α_1 — любые комплексные числа, сумма квадратов модулей которых равна единице: $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Например, возможно состояние $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, а также состояние $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Коэффициенты α_0 и α_1 называют *амплитудами* (amplitudes) состояний $|0\rangle$ и $|1\rangle$ соответственно. Напомним, что амплитуды могут быть комплексными, так что, скажем, можно рассмотреть состояние $\frac{1}{\sqrt{5}}|0\rangle + \frac{2i}{\sqrt{5}}|1\rangle$ (здесь i обозначает, как обычно, мнимую единицу, $i = \sqrt{-1}$). Такого рода суперпозиции, $\alpha_0|0\rangle + \alpha_1|1\rangle$, описывают состояние элементарной ячейки для хранения информации (см. рисунок 10.1), называемой *кубитом* (qubit).

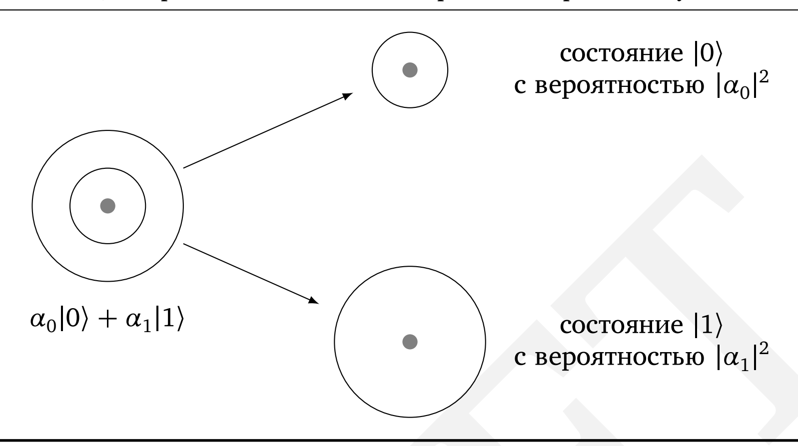
Рис. 10.1. Электрон в атоме водорода может находиться в основном и возбуждённом состояниях. В обозначениях Дирака, традиционно используемых в квантовой механике, их записывают как $|0\rangle$ и $|1\rangle$. Но *принцип суперпозиции* (superposition principle) говорит нам, что на самом деле возможны и *линейные комбинации* (linear combinations) этих состояний: $\alpha_0|0\rangle + \alpha_1|1\rangle$. Говорят, что состояние $\alpha_0|0\rangle + \alpha_1|1\rangle$ является суперпозицией *базисных состояний* $|0\rangle$ и $|1\rangle$. Если бы α_0 и α_1 были неотрицательными числами, в сумме равными 1, то это можно было бы понимать так: электрон находится в состоянии 0 с вероятностью $|\alpha_0\rangle$ и в состоянии 1 с вероятностью $|\alpha_1\rangle$. Однако α_0 и α_1 могут быть комплексными числами, и единице должна быть равна не их сумма, а сумма квадратов их модулей!



Выражаясь фигурально, можно сказать, что электрон в атоме ещё сам не решил, в каком состоянии он находится — основном или возбуждённом, а числа α_0 и α_1 выражают его «предрасположенность» к тому и другому состоянию. Но понимать «предрасположенность» как вероятность нельзя: α_0 и α_1 могут быть отрицательными или вообще мнимыми. И как тогда её понимать, спрашивается? Это одно из самых загадочных мест в квантовой механике, к которому наша интуиция приспосабливается с трудом.

Продолжая метафору, можно сказать, что состояние электрона (суперпозиция основного и возбуждённого состояний) описывает его «внутренний мир». Этот «внутренний мир» нам недоступен, но мы можем выполнить *измерение* (measurement), см. рис. 10.2. Результатом измерения является нуль или единица. При этом вероятности появления нуля и единицы равны со-

Рис. 10.2. Измерение системы в состоянии суперпозиции переводит её в одно из базисных состояний, и вероятности их появления равны квадратам модулей амплитуд.



ответственно $|\alpha_0|^2$ и $|\alpha_1|^2$ (напомним, что как раз сумма квадратов модулей должна быть равна единице). Побочным (но неизбежным) результатом измерения является то, что система оказывается в измеренном состоянии: если измерение дало 0, то система находится после измерения в основном состоянии ($|0\rangle = 1 \cdot |0\rangle + 0 \cdot |1\rangle$), а если 1 — то в возбуждённом ($|1\rangle$). Невозможность измерить состояние системы, не повлияв на него, — другая странная особенность квантовой механики, не имеющая аналогов в классической физике.

Принцип суперпозиции применим не только к системам с двумя состояниями. На самом деле даже в нашем примере (электрон в атоме водорода) возможно несколько возбуждённых состояний, и можно рассмотреть систему с k базисными состояниями: основным и $k-1$ возбуждёнными. Обозначим их $|0\rangle, |1\rangle, \ldots, |k-1\rangle$. Произвольная суперпозиция такой системы записывается тогда как $\alpha_0|0\rangle + \alpha_1|1\rangle + \ldots + \alpha_{k-1}|k-1\rangle$, где $|\alpha_0|^2 + |\alpha_1|^2 + \ldots + |\alpha_{k-1}|^2 = 1$. Результатом измерения в такой системе будет число от 0 до $k-1$, и вероятность появления числа j равна $|\alpha_j|^2$. Как мы уже говорили, измерение меняет состояние системы, и её новое состояние определяется результатом измерения.

Для кодирования n битов можно было бы использовать систему с 2^n базисными состояниями (скажем, достаточно возбуждённый атом водорода), но лучше использовать n кубитов.

Рассмотрим два кубита, представляющих собой состояния электронов в двух атомах водорода (в каждом атоме используются два состояния — основное и возбуждённое). В классической физике два состояния для каждого атома дают всего четыре комбинации 00, 01, 10, 11 (сначала мы записываем состояние одного атома, а потом — второго). Тем самым два атома можно использовать для хранения двух битов информации — по одному на атом. В квантовой механике принцип суперпозиции позволяет рассмотреть линей-

ную комбинацию четырёх основных состояний,

$$|\alpha\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle,$$

где

$$\sum_{x\in\{0,1\}^2} |\alpha_x|^2 = 1.$$

(Напомним, что $\{0, 1\}^2$ обозначает множество двоичных слов длины 2, и вообще $\{0, 1\}^n$ есть множество двоичных слов длины n.) Результатом измерения в такой системе будут два бита, и вероятность исхода $x \in \{0, 1\}^2$ равна $|\alpha_x|^2$. Если измерение дало исход $x = jk$, то после измерения система окажется в *чистом состоянии* x: если, скажем, $x = 10$, то первый электрон будет в возбуждённом состоянии, а второй — в основном.

Сцепленность

Пусть у нас есть два кубита. Первый находится в состоянии $\alpha_0|0\rangle + \alpha_1|1\rangle$, а второй — в состоянии $\beta_0|0\rangle + \beta_1|1\rangle$. Каково состояние системы, состоящей из этих двух кубитов? Ответ: тензорное произведение указанных выше состояний, которое имеет вид

$$\alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle.$$

Можно ли для произвольного состояния системы из двух кубитов выделить состояния каждого из них, обратив эту процедуру? Не всегда. Говорят, что кубиты *сцеплены* (entangled), если их совместное состояние невозможно разложить в произведение индивидуальных состояний. Рассмотрим, например, состояние $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ (одно из *состояний Белла*). Его невозможно разложить в произведение состояний первого и второго кубитов (см. упражнение 10.1). Сцепленность (entanglement) — главная загадка квантовой механики; именно на ней основана надежда на вычислительные возможности квантовых устройств.

Ещё одна интересная возможность — это возможность *частичного измерения* (partial measurement). Мы можем измерить, скажем, только первый бит. Какова вероятность, что мы обнаружим его в состоянии 0? Такая же, как если бы мы измерили два бита сразу, то есть $|\alpha_{00}|^2 + |\alpha_{01}|^2$. Более сложный вопрос: в каком состоянии окажется система после такого измерения?

Квантовая механика отвечает на этот вопрос так: если результат измерения первого бита равен 0, то в суперпозиции останутся только члены с таким значением первого бита, а коэффициенты перед ними будут перенормированы: умножены на положительное число с тем расчётом, чтобы сумма квадратов снова стала равной единице. В нашем примере новым состоянием будет

$$|\alpha_{\text{новое}}\rangle = \frac{\alpha_{00}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}|00\rangle + \frac{\alpha_{01}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}|01\rangle.$$

Теперь рассмотрим случай n атомов (в качестве n можно взять сравнительно небольшое число, скажем, $n = 500$). С классической точки зрения, такая система может хранить 500 битов информации. В квантовой механике её состояние представляет собой линейную комбинацию 2^{500} классических (базисных) состояний с комплексными коэффициентами:

$$\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$$

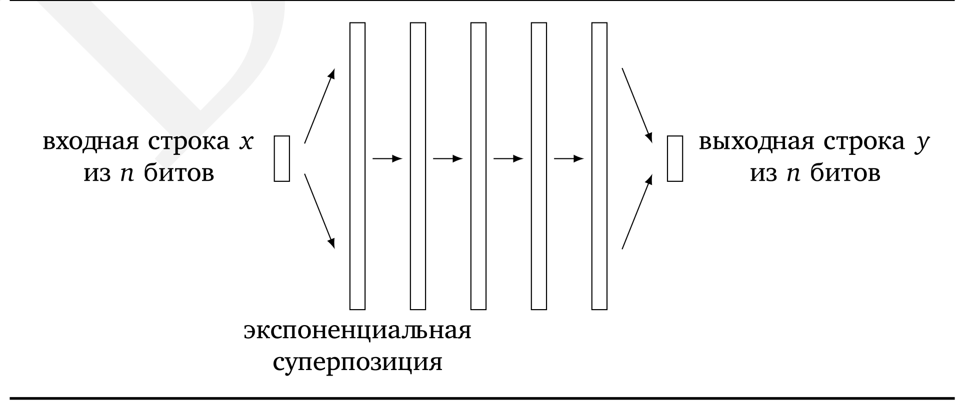
Получается, что Природе нужно помнить 2^{500} комплексных чисел — и это только для состояния 500 атомов водорода! Более того, эти числа надо не только помнить, но и менять, если состояние системы меняется со временем.

Число 2^{500} много больше числа элементарных частиц во всей Вселенной. Неужели Природа может хранить столько информации? И где? И это только для одной совсем небольшой системы. Как ни странно, именно так всё и происходит согласно квантовой механике.

Раз так, почему бы не обратить это себе на пользу и не использовать возможности Природы для вычислений? В этом и состоит основная идея квантовых компьютеров.

Реализовать её, однако, совсем не просто. Одна из трудностей: упомянутая сумма экспоненциального размера отражает «внутренний мир» системы, а в процессе измерения мы получаем всего лишь n битов информации. (Напомним, что вероятность получить в результате измерения слово x длины n равна $|\alpha_x|^2$, квадрату модуля соответствующего коэффициента, и после измерения система переходит в состояние $|x\rangle$.)

Рис. 10.3. Квантовый алгоритм может взять n классических битов входа, сделать их состояниями n кубитов, затем произвести какие-то действия и получить в результате суперпозицию состояний с 2^n коэффициентами, после чего в процессе измерения получить снова n классических битов (в соответствии с распределением вероятностей, задаваемым квадратами модулей коэффициентов). Каждое из действий с суперпозицией воздействует на экспоненциальное число коэффициентов — и тем не менее с физической точки зрения остаётся всего лишь одним шагом.



10.2. План действий

Квантовый алгоритм не будет похож на классические. Причина этого в необходимости всё время иметь в виду разницу между внутренним состоянием экспоненциального размера и доступной для наблюдения информацией (всего лишь n битов).

На входе и выходе квантового алгоритма мы имеем слова из n (классических) битов; работа с экспоненциальными объёмами информации, которую Природа для нас выполняет по законам квантовой механики, остаётся ненаблюдаемой.

Получив на вход слово из n битов, мы приводим систему в состояние $|x\rangle$. Затем выполняются какие-то операции (по законам квантовой механики), и в результате наша система из n кубитов приходит в состояние $\sum_y \alpha_y |y\rangle$. (Мы предполагаем для простоты, что число кубитов в квантовом компьютере равно размеру входа, что, конечно, вовсе не обязательно.) Наконец, мы выполняем измерение и получаем на выходе некоторое n-битовое слово y с вероятностью $|\alpha_y|^2$.

То, что выходное слово алгоритма случайно, само по себе не страшно: с такой ситуацией мы уже встречались, говоря о вероятностных алгоритмах (скажем, алгоритме проверки простоты). Нужно только, чтобы слово y с большой вероятностью содержало в себе информацию о верном ответе. Тогда, повторив вычисление несколько раз, мы можем сделать вероятность ошибки пренебрежимо малой.

Обсудим теперь более подробно квантовую часть алгоритма разложения на множители. Как мы вскоре увидим, с помощью квантовых операций мы сможем искать некоторые закономерности в коэффициентах суперпозиции (периодичность), и информация об этих закономерностях поможет разложить число на множители. Но до этого мы должны будем преобразовать («распаковать») n классических битов входа в экспоненциальную сумму таким образом, чтобы эти закономерности в коэффициентах появились. После этого другие операции (и заключительное измерение) позволят их найти.

Всё это звучит более чем загадочно, но вскоре мы все эти шаги объясним. В разделе 10.3 мы описываем основной механизм, используемый в алгоритме — квантовый вариант быстрого преобразования Фурье (FFT). Затем мы опишем закономерности, которые с его помощью можно найти, и сведём задачу разложения на множители к поиску таких закономерностей. Наконец, мы покажем, как на первой стадии преобразовать заданное нам число N в суперпозицию экспоненциального размера, в которой эти закономерности следует искать.

Вот схема алгоритма разложения N на множители с помощью квантового компьютера (выделенные курсивом термины будут объяснены дальше):

- Разложение на множители сводится к нахождению *нетривиального квадратного корня из единицы* по модулю N.
- Нахождение такого корня сводится к вычислению *порядка* случайного числа по модулю N.

- Порядок числа по модулю N оказывается *периодом* некоторой *периодической* суперпозиции.
- Наконец, периоды суперпозиций можно вычислять с помощью *квантового преобразования Фурье*.

  Мы начнём с последнего шага.

10.3. Квантовое преобразование Фурье

Как мы видели в главе 2, входом преобразования Фурье является M-мерный комплексный вектор α, а результатом преобразования является вектор β, для которого

$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{M-1} \end{bmatrix} = \frac{1}{\sqrt{M}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{M-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(M-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(M-1)j} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{(M-1)} & \omega^{2(M-1)} & \dots & \omega^{(M-1)(M-1)} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{M-1} \end{bmatrix}.$$

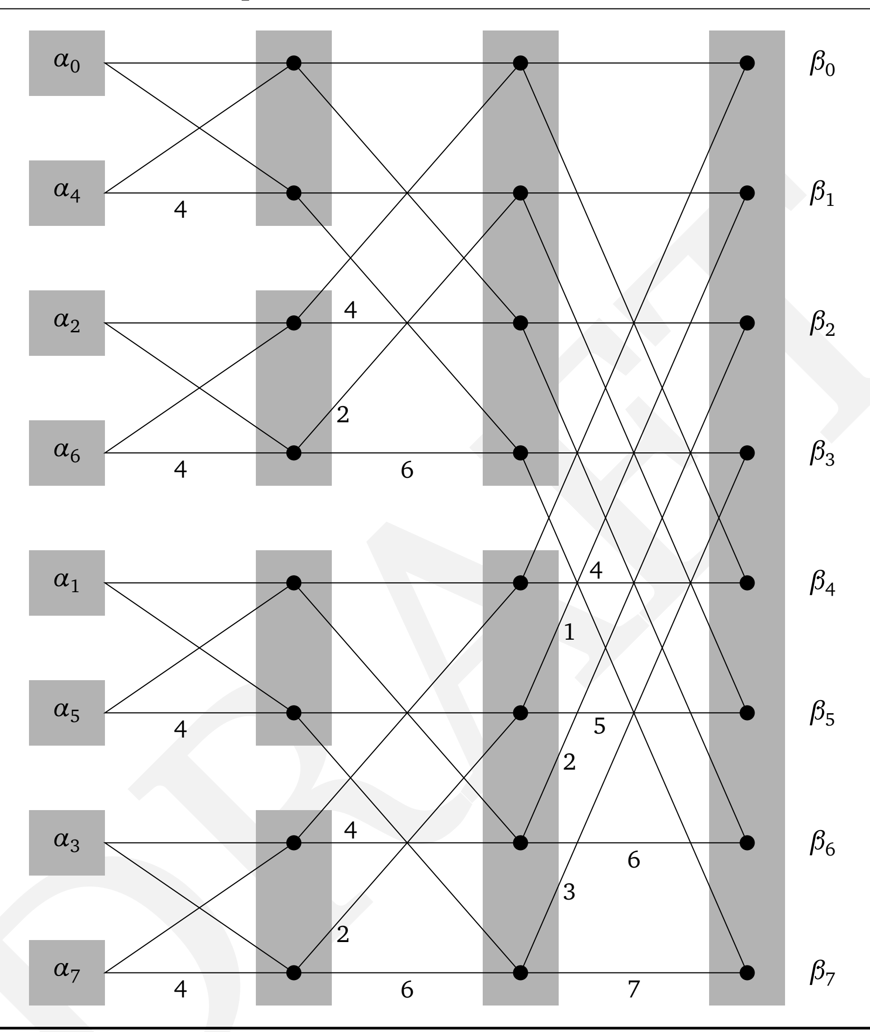
Здесь ω — корень из единицы степени M (комплексный). По сравнению с главой 2 мы добавили множитель $\sqrt{M}$. Он нужен для того, чтобы сумма $|\beta_i|^2$ равнялась единице, если она равна единице для $|\alpha_i|^2$.

Непосредственное применение этой формулы требует $O(M^2)$ шагов. Но, как мы видели, если M есть степень двойки, есть более эффективный способ, требующий всего $O(M \log M)$ шагов. (Благодаря этому цифровая обработка сигналов становится реальной.) Мы сейчас увидим, что квантовые алгоритмы позволяют *экспоненциально* ускорить вычисления и выполнить преобразование Фурье за $O(\log^2 M)$ шагов.

Но, спрашивается, как же время работы алгоритма может быть меньше длины входа? Дело в том, что вход кодируется как коэффициенты суперпозиции $m = \log M$ кубитов (у такого состояния как раз $2^m = M$ коэффициентов). В наших обозначениях $|\alpha\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$, где α_j — амплитуда m-битового слова, являющего двоичной записью числа j. (Можно сказать, что вектора $|j\rangle$ образуют базис, а α_j — коэффициенты разложения в этом базисе.)

Квантовый алгоритм преобразования Фурье преобразует эту суперпозицию, и это требует $m = \log M$ этапов. После выполнения очередного этапа коэффициенты текущей суперпозиции представляют собой промежуточные результаты в схеме быстрого преобразования Фурье. (См. рисунок 10.4, взятый из главы 2.) Как мы увидим, каждый этап может быть реализован как последовательность $\log M$ элементарных операций. Таким образом, в общей сложности получается $\log^2 M$ операций, после которых коэффициенты суперпозиции хранят результат преобразования Фурье.

Рис. 10.4. Схема быстрого преобразования Фурье (классического) для векторов длины $M = 2^m$ содержит $m = \log_2 M$ уровней.



Такая быстрота выглядит заманчиво, но надо иметь в виду, что тут есть некоторый подвох. Классический алгоритм преобразования Фурье реально выдаёт все компоненты результирующего вектора одну за другой. Квантовый алгоритм вместо этого создаёт состояние, в котором они являются коэффициентами, и извлечь их (или хотя бы один коэффициент) нельзя.

Измеряя результат, мы получим в качестве ответа m-битовую строку, причём вероятность получить строку j равна $|\beta_j|^2$. Так что мы даже и один коэффициент не узнаём — а только узнаём один из индексов, где квадрат модуля коэффициента не слишком мал (этот квадрат — вероятность появления этого индекса).

Таким образом, было бы честнее говорить не о квантовом преобразовании Фурье, а о *квантовой выборке из преобразования Фурье*:

Вход: суперпозиция системы из $m = \log_2 M$ кубитов, $|\alpha\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$.

Действия: последовательность из $O(m^2) = O(\log^2 M)$ элементарных квантовых операций с системой, в результате которых она приходит в состояние $|\beta\rangle = \sum_{j=0}^{M-1} \beta_j |j\rangle$.

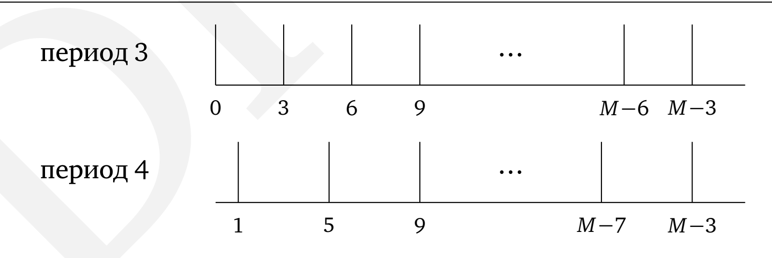
Выход: случайное m-битовое число j (другими словами, $0 \leqslant j \leqslant M-1$) по распределению вероятностей $\mathrm{P}(j) = |\beta_j|^2$.

Можно сказать, что квантовое преобразование Фурье — это быстрый способ получить хоть какую-то информацию о результате преобразования Фурье — а именно, узнать какой-то из его не очень малых коэффициентов. Точнее, мы узнаём не сам коэффициент, а индекс. С первого взгляда трудно себе представить, как можно извлечь пользу из такой скудной информации. Оказывается, что иногда можно — и мы сейчас обсудим, как.

10.4. Периодичность

Предположим, что последовательность $\alpha_0, \alpha_1, \ldots, \alpha_{M-1}$ имеет период k, где k — делитель M. Тогда $\alpha_i = \alpha_j$ при $i \equiv j \pmod{k}$, и последовательность разбивается на M/k одинаковых участков длины k. Будем говорить, что состояние $|\alpha\rangle$ периодично *с периодом k и сдвигом j*, если в каждом из участков только один член отличен от нуля, и это $\alpha_j, \alpha_{j+k}, \alpha_{j+2k}, \ldots$

Рис. 10.5. Примеры периодических состояний (показаны ненулевые коэффициенты).



Для такого состояния выборка из преобразования Фурье (повторённая несколько раз) позволяет с большой вероятностью найти его период. В самом деле, как мы увидим (см. врезку), верно такое утверждение:

Пусть входом квантового алгоритма выборки из преобразования Фурье является состояние с периодом k и сдвигом j, где k — некоторый делитель M. Тогда индекс на выходе будет кратен M/k, и все кратные M/k одинаково вероятны.

Преобразование Фурье периодического вектора

Рассмотрим вектор $|\alpha\rangle$, который имеет период k и сдвиг 0 (с ненулевыми членами $\alpha_0, \alpha_k, \alpha_{2k}, \ldots$, всего M/k штук). Чтобы сумма квадратов всех модулей ненулевых членов равнялась 1, сделаем их равными $\sqrt{k/M}$:

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}}|jk\rangle.$$

В этом случае преобразование Фурье также будет периодическим с периодом M/k и сдвигом 0:

$$|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \left| \frac{jM}{k} \right\rangle.$$

Доказательство. В сумме, определяющей преобразование Фурье, остаются только члены с номерами, кратными k:

$$\beta_j = \frac{1}{\sqrt{M}} \sum_{l=0}^{M-1} \omega^{jl}\alpha_l = \frac{\sqrt{k}}{M} \sum_{i=0}^{M/k-1} \omega^{jik}.$$

В правой части стоит геометрическая прогрессия из M/k членов со знаменателем ω^{jk}. Если jk кратно M, то знаменатель равен 1, так как ω — комплексный корень из единицы степени M. Тогда сумма прогрессии равна M/k, так что $\beta_j = 1/\sqrt{k}$ при этих j (то есть при j, кратных M/k). Если же jk не кратно M, то можно воспользоваться формулой для суммы геометрической прогрессии, $(1-\omega^{jk(M/k)})/(1-\omega^{jk})$, в которой теперь знаменатель не равен нулю, а числитель равен нулю (так как $jk(M/k) = jM$ кратно M, а $\omega^M = 1$).

В итоге $\beta_j = 1/\sqrt{k}$ при j, кратных M/k, и $\beta_j = 0$ при остальных j. □

Если же исходный вектор имеет период k со сдвигом $l < k$, то есть

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}}|jk+l\rangle,$$

то преобразование Фурье β по-прежнему будет сосредоточено в кратных M/k:

$$|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \omega^{ljM/k} \left| \frac{jM}{k} \right\rangle.$$

Это доказывается примерно таким же вычислением, как и раньше (упражнение 10.5 показывает, что сдвиг в аргументе соответствует умножению компонент результата на подходящие множители).

Отсюда мы заключаем (по линейности), что у любого состояния с периодом k преобразование Фурье сосредоточено в координатах, кратных M/k. Если к тому же лишь один коэффициент в периоде исходного состояния от-

личен от 0, то ненулевые коэффициенты в преобразовании Фурье имеют равные модули, и последующая выборка даст нам одно из k таких кратных с вероятностью $1/k$.

Теперь мы можем выполнить описанную процедуру несколько раз и взять наибольший общий делитель полученных индексов. Несложно сообразить, что с большой вероятностью этот делитель будет равен M/k, и тем самым мы сможем найти период k входной последовательности!

Оценим вероятность получения правильного наибольшего общего делителя описанным способом.

Лемма. *Рассмотрим s чисел, независимо выбранных из*

$$0, \frac{M}{k}, \frac{2M}{k}, \ldots, \frac{(k-1)M}{k}$$

(*все k значений равновероятны*). *Тогда с вероятностью как минимум $1 - k/2^s$ наибольший общий делитель выбранных чисел равен M/k.*

Доказательство. Это может быть не так, лишь если все выбранные числа кратны некоторому $j \cdot M/k$ при некотором j от 2 до k. Вероятность этого (для фиксированного j) не превосходит $1/j \leqslant 1/2$ при одном испытании и тем самым $1/2^s$ при s испытаниях.

Теперь надо сложить вероятности для разных j и получить оценку для искомой вероятности (вероятность того, что произойдёт одно из событий, не больше суммы их вероятностей). В итоге получаем, что вероятность нежелательного исхода не больше $k/2^s$. □

Эту вероятность можно сделать сколь угодно малой, если s взять достаточно большим.

10.5. Квантовые схемы

Мы упоминали, что квантовые схемы могут вычислить преобразование Фурье — но не объяснили, как это делается (и даже что это значит). Что представляют собой *квантовые схемы*, о которых идёт речь, и как с их помощью быстро вычислить преобразование Фурье?

10.5.1. Элементы квантовых схем

В классическом случае схемы строят из функциональных элементов (AND, NOT и т. п.), имеющих один или два входа. Аналогичным образом мы будем рассматривать квантовые операции с одним или двумя кубитами. Рассмотрим преобразование Адамара H, которое преобразует состояния $|0\rangle$ и $|1\rangle$ так:

$$|0\rangle \longrightarrow \boxed{H} \longrightarrow \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \qquad |1\rangle \longrightarrow \boxed{H} \longrightarrow \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.$$

Если в любом из получившихся состояний произвести измерение, то оно даст 0 или 1 с равными вероятностями. Но мы можем применить элемент Адамара и ко входу в состоянии суперпозиции $\alpha_0|0\rangle + \alpha_1|1\rangle$, и результат определяется по линейности:

$$H(\alpha_0|0\rangle + \alpha_1|1\rangle) = \alpha_0 H(|0\rangle) + \alpha_1 H(|1\rangle) = \frac{\alpha_0 + \alpha_1}{\sqrt{2}}|0\rangle + \frac{\alpha_0 - \alpha_1}{\sqrt{2}}|1\rangle.$$

По этой формуле легко проверить, что двукратное применение преобразования Адамара возвращает кубит в исходное состояние.

Другой базовый элемент — управляемое отрицание (controlled NOT, или CNOT). Он применяется к двум кубитам. Первый (управляющий) кубит говорит, нужно ли обращать второй (но сам не меняется). Если первый равен нулю, то второй остаётся без изменения, если равен единице — то обращается. Скажем, CNOT$(|00\rangle) = |00\rangle$ и CNOT$(|10\rangle) = |11\rangle$.

$|00\rangle$ → $|00\rangle$ $\qquad$ $|10\rangle$ → $|11\rangle$

В отличие от элемента Адамара, элемент CNOT преобразует базисные состояния в базисные, так что он задаёт некоторую перестановку 4 классических состояний системы из двух битов.

Другой квантовый элемент (управляемый сдвиг фазы) мы опишем в дальнейшем, когда он нам понадобится.

Мы описали действие элементов на состояния системы из одного или двух кубитов. Но можно применить этот элемент и к одному (или двум) кубитам системы из n кубитов. Квантовая механика позволяет сказать, как при этом изменятся все 2^n коэффициентов системы. Пусть, скажем, система находилась в состоянии $\alpha = \sum_{x\in\{0,1\}^n} \alpha_x|x\rangle$, и мы применили элемент Адамара к первому кубиту. Тогда новое состояние $\beta = \sum_{x\in\{0,1\}^n} \beta_x|x\rangle$, согласно правилам квантовой механики, будет иметь коэффициенты $\beta_{0y} = \frac{\alpha_{0y} + \alpha_{1y}}{\sqrt{2}}$ и $\beta_{1y} = \frac{\alpha_{0y} - \alpha_{1y}}{\sqrt{2}}$ для любого $(n-1)$-битового суффикса y. Другими словами, элемент Адамара применяется параллельно к 2^{n-1} группам из двух коэффициентов (для каждого y своя группа). Именно это и позволяет квантовым алгоритмам работать так быстро (в частности, при вычислении преобразования Фурье).

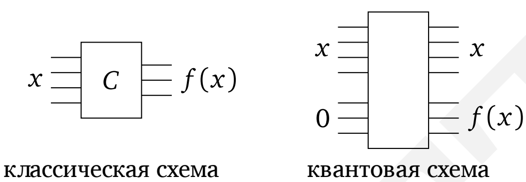
10.5.2. Две основные квантовые схемы

На рисунке, изображающем квантовую схему, начальные состояния входных n кубитов изображаются проводами (слева). Каждый квантовый элемент подвергает преобразованию входящие в него слева кубиты, в результате чего состояние всей системы изменяется, и с правой стороны рисунка провода обозначают результирующее состояние кубитов после всех преобразований.

На верхнем уровне нам понадобятся две конструкции:

Квантовое преобразование Фурье получает n кубитов в некотором состоянии $|\alpha\rangle$ и выдаёт состояние $|\beta\rangle$, коэффициенты которого получаются преобразованием Фурье из коэффициентов $|\alpha\rangle$.

Классические вычисления могут быть преобразованы в квантовые в следующем смысле. Пусть есть (классическая) функция, преобразующая n-битовые слова в m-битовые. Соответствующая (классическая) схема имеет n входов и m выходов. Тогда можно построить квантовую схему, имеющую $n + m$ входов и выходов, с таким свойством. Если подать на вход некоторое битовое слово x и m нулей, то на выходе мы получим x и значение $f(x)$ (соответственно на первой и второй группе выходов).



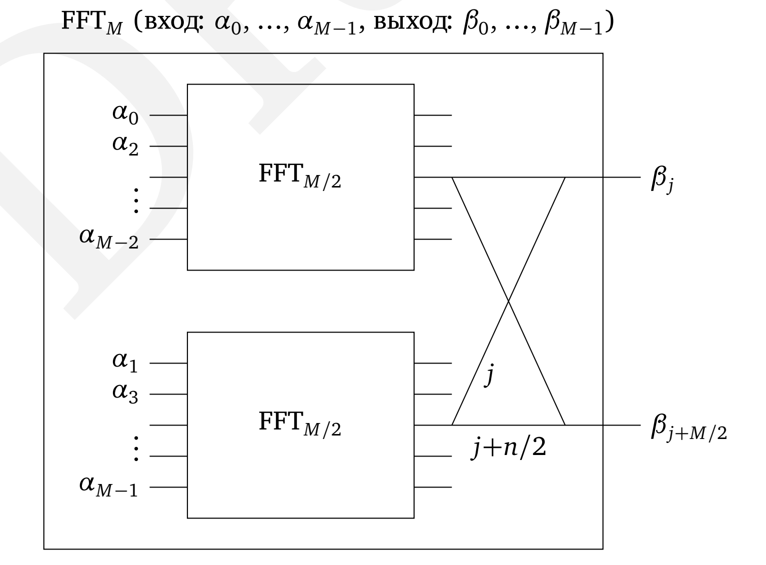
Заметим, что пока речь шла о преобразовании базисных состояний. Линейность гарантирует, что если подать на вход такой схемы суперпозицию базисных состояний $|x, 0^k\rangle$ с какими-то коэффициентами, то получится сумма состояний $|x, f(x)\rangle$ с теми же коэффициентами.

В принципе можно принять на веру возможность построения таких схем и исходя из этого понять работу алгоритма разложения на множители. Однако для более дотошных читателей мы опишем подробнее устройство квантового преобразования Фурье в следующем разделе. О построении квантовых аналогов классических схем см. упражнение 10.7.

10.5.3. Схема квантового преобразования Фурье

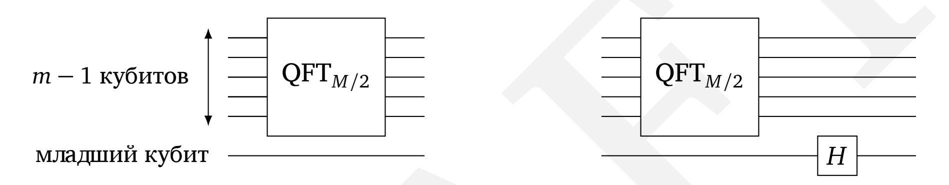
Вспомним, как мы в разделе 2.6.4 сводили вычисление преобразования Фурье вектора длины M к двум преобразованиям векторов половинного размера.

FFT_M (вход: $\alpha_0, \ldots, \alpha_{M-1}$, выход: $\beta_0, \ldots, \beta_{M-1}$)



Посмотрим, во что это превращается в квантовом случае. Вход теперь представляет собой вектор из $M = 2^m$ амплитуд системы из m кубитов. Разделение его компонент на чётные и нечётные происходит по значению одного из кубитов (младшего из m разрядов). Таким образом, мы должны рекурсивно применить преобразование Фурье меньшего размера отдельно к коэффициентам при тех состояниях, где младший кубит равен нулю, и отдельно — при тех, где младший кубит равен единице.

Ключевое обстоятельство: для этого достаточно применить схему преобразования Фурье меньшего размера к остальным $m-1$ кубитам. При этом необходимые действия будут автоматически выполнены параллельно для одной и другой группы. Тем самым вместо двух задач меньшего размера (как в классическом случае) мы получаем только одну — и за счёт этого квантовый алгоритм экспоненциально быстрее классического.



Но это ещё не совсем всё: результаты выполнения преобразования для каждой половины нужно смешать с подходящими коэффициентами. Если вспомнить соответствующую формулу, то видно, что коэффициенты с индексами j и $M/2+j$ надо сложить (для получения j-го коэффициента выходного вектора) и вычесть (для $(M/2+j)$-го). Точнее, коэффициент с индексом $M/2+j$ надо предварительно «сдвинуть по фазе», умножив на подходящую степень ω.

Если бы этого сдвига не было, то нужное преобразование — одновременно для всех коэффициентов — можно было бы выполнить, применив элемент Адамара к старшему кубиту. Мы уже говорили, что такое преобразование параллельно комбинирует коэффициенты с индексами $0x$ и $1x$, что как раз (если перейти от двоичных слов к числам) соответствует индексам x и $M/2+x$. Так что и в этой части — как это ни удивительно на первый взгляд — достаточно всего одного квантового элемента.

Мы пока не объяснили, как обеспечить в промежутке между этими двумя действиями необходимый сдвиг фазы. Если посмотреть на формулы, то видно, что это нужно сделать только для половины коэффициентов (там, где старший бит 1) и что коэффициент с номером $M/2+j$ надо умножить на ω^j. Умножение на $1, \omega, \omega^2, \ldots, \omega^{M/2-1}$ можно реализовать в несколько шагов: сначала умножить на ω те коэффициенты, где требуется нечётная степень j (равен единице младший бит индекса), затем умножить на ω^2 те коэффициенты, где второй справа бит индекса равен единице, и так далее. Если разобраться, то видно, что это можно сделать с помощью нескольких операций условного сдвига фазы — операции над двумя кубитами, которая оставляет $|00\rangle$, $|01\rangle$ и $|10\rangle$ без изменения, а $|11\rangle$ умножает на некоторое комплексное число, по модулю равное единице (ω, ω^2, ω^4 и т. д.), — матрица такого пре-

образования диагональна, и на диагонали стоят три единицы и одно число, равное единице по модулю. (Напомним, что нам нужно умножать лишь коэффициенты, у которых старший бит и ещё один бит равны единице.) Всё это (кого и на что надо умножать и как складывать) удобно проследить на рис. 10.4; заметим ещё, что перестановка квантовых битов не представляет сложности, так как мы разрешаем выполнять операции с любыми парами кубитов.

Итак, схема квантового преобразования Фурье описана. Каждый из m уровней рекурсии в ней требует $O(m) = O(\log M)$ операций, всего получается $O(m^2)$, то есть $O(\log^2 M)$ квантовых операций.

10.6. Периодичность и разложение на множители

Мы видели, как можно использовать преобразование Фурье для отыскания периода. Покажем теперь, как (за несколько простых шагов) можно свести разложение на множители к отысканию периода.

Пусть дано некоторое целое положительное N. *Нетривиальным квадратным корнем из N* мы будем называть число x, для которого $x^2 \equiv 1 \pmod N$, но $x \not\equiv \pm 1 \pmod N$. (Ср. с упражнениями 1.36 и 1.40.) Зная нетривиальный квадратный корень, легко разложить N на два множителя, не равных единице (и эти множители можно разлагать дальше, пока не дойдём до простых). Вот как это делается.

Лемма. *Пусть x — нетривиальный квадратный корень из N. Тогда число* НОД$(x+1, N)$ *является делителем N, отличным от* 1 *и* N.

Доказательство. Поскольку $x^2 \equiv 1 \pmod N$, то $x^2 - 1 = (x-1)(x+1)$ делится на N. Но ни один из сомножителей $(x-1)$ и $(x+1)$ по условию на N не делится, поэтому некоторые простые множители N должны войти в один сомножитель, а другие — в другой. В такой ситуации НОД$(x+1, N)$ (а также НОД$(x-1, N)$) будет содержать только некоторые сомножители N. □

Пример. Пусть $N = 15$. Тогда число 4 является нетривиальным квадратным корнем из N, поскольку $4^2 = 16 \equiv 1 \pmod{15}$, но $4 \not\equiv \pm 1 \pmod{15}$. Как и предсказывает лемма, НОД$(4-1, 15) = 3$ и НОД$(4+1, 15) = 5$ оказываются нетривиальными делителями 15.

Нам понадобится ещё одно понятие, чтобы установить обещанную связь с периодичностью. *Порядком* числа x по модулю N называется наименьшее положительное целое число r, при котором $x^r \equiv 1 \pmod N$. Например, порядок числа 2 по модулю 15 равен 4.

Задача отыскания порядка случайного остатка по модулю N, как мы увидим, связана с отысканием нетривиального квадратного корня из N (и тем самым с разложением на множители). Это видно из следующей леммы.

Лемма. *Пусть N — нечётное составное число, имеющее как минимум два разных простых множителя. Выберем случайное число x от* 0 *до* $N-1$*, взаимно простое с N, считая все такие числа равновероятными. Тогда с вероятно-*

стью $1/2$ *или больше порядок* r *числа* x *по модулю* N *окажется чётным, а число* $x^{r/2}$ *будет нетривиальным квадратным корнем из* N.

Доказательство этой леммы мы оставляем читателю в качестве упражнения. Она показывает, что можно разлагать число N на множители, умея вычислять порядок. Достаточно взять случайный остаток x по модулю N. Если он не взаимно прост с N, что можно проверить по НОД(x, N), то мы сразу получаем разложение на два множителя, один из которых НОД(x, N). Если же x взаимно прост с N, то вычисляем порядок — с вероятностью $1/2$ или более он окажется чётным и даст нетривиальный квадратный корень. Тем самым мы сможем разложить N на два множителя.

Пример. При $x = 2$ и $N = 15$ порядок оказывается чётным (он равен 4, и $2^4 = 16 \equiv 1 \pmod{15}$). Возводя x в половинную степень, получаем $x^2 = 4$, и это будет нетривиальный корень из 15, по которому можно найти делитель N, например, НОД$(x + 1, N) = 5$.

Таким образом, мы свели задачу разложения на множители (factoring) к задаче отыскания порядка остатка по данному модулю (order finding). Преимущество второй задачи в том, что она связана с естественной периодической функцией. При данных N и x мы можем рассмотреть функцию $a \mapsto f(a) = x^a \bmod N$. Если r — порядок x по модулю N, то r будет периодом последовательности значений функции f:

$$f(0) = f(r) = f(2r) = \ldots = 1, \quad f(1) = f(r+1) = f(2r+1) = \ldots = x$$

и так далее. Заметим, что функцию f можно быстро вычислять с помощью повторного возведения в квадрат (алгоритм раздела 1.2.2). Таким образом, осталось понять (для разложения N на множители), как можно с помощью функции f организовать суперпозицию, коэффициенты которой будут иметь период r; после этого мы можем с помощью повторной выборки из квантового преобразования Фурье с большой вероятностью найти это r и затем использовать его для отыскания нетривиального корня из N и тем самым нетривиального множителя. Как это делается, объяснено ниже (см. врезку).

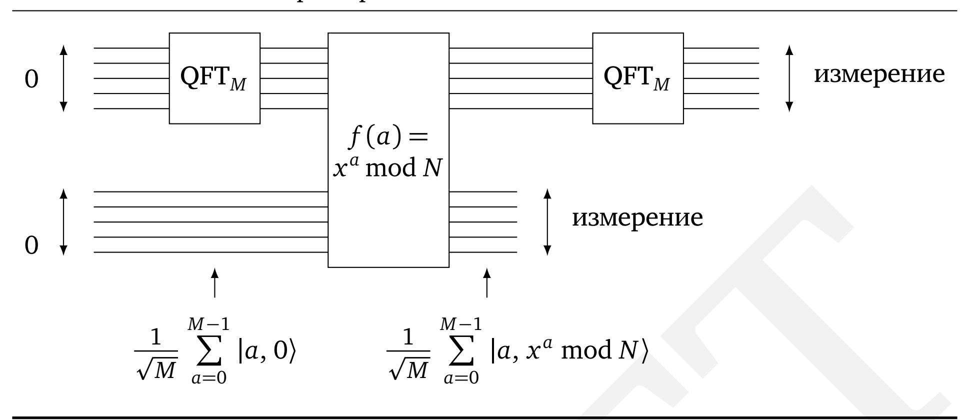
О превращении классической схемы в квантовую подробнее см. в упражнении 10.7.

10.7. Квантовый алгоритм разложения на множители

Теперь мы можем собрать вместе описанные выше компоненты квантового алгоритма разложения на множители (см. рисунок 10.6). Поскольку за полиномиальное время можно проверить, является ли вход простым числом или степенью простого числа, будем считать, что это уже сделано. Достаточно рассмотреть случай, когда данное нам число N является нечётным составным числом, имеющим как минимум два различных простых множителя.

Вход: составное нечётное число N (имеющее как минимум два простых множителя).
Выход: один из множителей числа N.

Рис. 10.6. Квантовый алгоритм разложения на множители.



1. Выбираем случайное (равномерно распределённое) число x между 1 и $N-1$. (Если НОД$(x,N)\neq 1$, то множитель уже найден, так что считаем x взаимно простым с N.)

2. В качестве M выберем степень двойки, близкую к N. (Точнее говоря, лучше всего выбрать $M\approx N^2$, но объяснение этого выходит за рамки нашего рассказа.)

3. Повторяем $s=2\log N$ раз такие действия:

a. Слова из нулевых битов помещаем в два квантовых регистра, первый из которых вмещает в себя остаток по модулю M, а второй — по модулю N.

b. Используем периодическую функцию $f(a)=x^a \bmod N$ для создания периодической суперпозиции длины M (см. соответствующую врезку):

(i) Применяем квантовое преобразование Фурье к первому регистру и получаем в нём суперпозицию $\sum_{a=0}^{M-1}\frac{1}{\sqrt{M}}|a,0\rangle$.

(ii) Вычисляем $f(a)=x^a \bmod N$ с использованием квантовой схемы, получаем суперпозицию $\sum_{a=0}^{M-1}\frac{1}{\sqrt{M}}|a,x^a \bmod N\rangle$.

(iii) Измеряем содержимое второго регистра. После этого первый регистр содержит периодическую суперпозицию $|\alpha\rangle=\sum_{j=0}^{M/r-1}\sqrt{\frac{r}{M}}|jr+k\rangle$, где k — некоторый сдвиг между 0 и $r-1$, а r — период x по модулю N.

c. Производим квантовую выборку из преобразования Фурье первого регистра, получая в результате некоторый индекс между 0 и $M-1$. (То, что у нас есть ещё и второй регистр, не помеха — просто ко всем состояниям добавляется постоянное значение во втором регистре.)

Пусть g — наибольший общий делитель полученных индексов $j_1,\ldots,j_s$.

4. Если M/g чётно, то вычисляем НОД$(N,x^{M/2g}+1)$. Если мы получили нетривиальный делитель N, выдаём его в качестве результата, если нет — повторяем выполнение алгоритма, начиная с шага 1.

Суперпозиция с периодическими коэффициентами

Покажем, как использовать периодичность функции $f(a) = x^a \bmod N$ для получения суперпозиции с периодическими коэффициентами.

- Возьмём два квантовых регистра (набора кубитов), в которых изначально все кубиты находятся в нулевом состоянии.
- Применим квантовое преобразование Фурье к первому регистру (состоящему из $m = \log M$ кубитов). В результате в нём появится суперпозиция всех значений от 0 до $M - 1$ с коэффициентами $1/\sqrt{M}$ при каждом из них. (В самом деле, вначале коэффициент при $|00\ldots0\rangle$ был равен единице, а остальные — нулю, остаётся вспомнить формулу, задающую преобразование Фурье.)
- Применяем квантовую схему, соответствующую классическому вычислению функции $f(a) = x^a \bmod N$. При этом в качестве входа используем первый регистр (где хранятся все строки длины M с равными амплитудами), а результат помещаем во второй регистр (изначально он содержал все нули с амплитудой 1). В результате получаем состояние двух регистров, равное $\sum\limits_{a=0}^{M-1} \frac{1}{\sqrt{M}}|a, f(a)\rangle$.
- Производим измерение второго регистра. В результате его значение будет фиксировано (тем или иным способом), а в первом регистре окажется суперпозиция состояний с периодическими амплитудами.

  В самом деле, раз функция f имеет период r, то значением второго регистра после измерения будет одно из чисел $f(k)$ при k от 0 до $M - 1$. А в первом регистре (как мы говорили выше) останутся с равными коэффициентами те значения a, которые совместимы с измеренным состоянием второго регистра. Поскольку в последовательности $f(0), f(1), f(2), \ldots$ повторения случаются только через r шагов, то в первом регистре останется периодическая последовательность (одинаковые коэффициенты, разделённые нулями), и квантовая выборка из преобразования Фурье позволит найти период r, что нам и требовалось.

Как мы видели (см. леммы выше), этот метод работает по крайней мере для половины возможных значений x, так что всю процедуру с большой вероятностью достаточно выполнить всего несколько раз.

Правда, в наших объяснениях есть существенная неувязка. Число M должно быть степенью двойки, чтобы мы смогли применить квантовое преобразование Фурье. С другой стороны, способ определения периода предполагает, что период является делителем M (то есть, выходит, степенью двойки — на что нет никаких оснований надеяться).

Оказывается, что эта трудность преодолима: *квантовое преобразование Фурье позволяет найти период вектора коэффициентов даже в том случае, если он не делит M*. Однако объяснить это так просто уже не получается, и мы не будем этого делать.

Квантовая физика и вычисления

Когда компьютеры только появились, не было очевидным, что построение схем из элементарных логических элементов является наиболее эффективным способом организации вычислений. Но к 1970-м годам сомнения в этом исчезли. Архитектура фон Неймана и кремниевые микросхемы явно победили, и все считали, что любой другой способ построения может дать максимум полиномиальный выигрыш. (Это означает, что T шагов вычисления на любом другом компьютере можно моделировать за полиномиально зависящее от T число шагов на имеющемся.) Этот тезис называют *расширенным тезисом Чёрча—Тьюринга*. Идея квантового компьютера противоречит этому тезису и потому ставит под сомнения наши базовые представления о возможностях компьютеров.

Реализуемы ли на самом деле квантовые компьютеры? На этот вопрос пытаются ответить и физики, и информатики во всём мире. Главная трудность состоит в том, что суперпозиции очень хрупки и требуют изоляции от непредвиденных взаимодействий (в частности, измерений) с окружением. Кое-какой прогресс тут происходит, но очень медленный. До сих пор наибольшим достижением было квантовое разложение на множители числа 15 (именно так!) с помощью ядерного магнитного резонанса (nuclear magnetic resonance, NMR). Да и то по этому поводу были вопросы — насколько тут квантовая механика была по существу.

Может быть, квантовые компьютеры так и не удастся построить? Может быть, это было бы даже более интересным результатом, если при этом удалось бы обнаружить дефекты самой квантовой механики — после столетней истории её развития.

Перспектива лучше понять законы природы на квантовом уровне ничуть не менее заманчива, чем надежда построить новые супербыстрые компьютеры...

Пусть входное число N содержит $n = \log N$ битов. Алгоритм требует $2 \log N = O(n)$ итераций (шага 3), и каждая из них (возведение в степень по модулю, раздел 2.6.4, отыскание наибольшего общего делителя с помощью алгоритма Евклида, раздел 1.2.3, квантовое преобразование Фурье) требует полиномиального (от n) числа шагов. Поэтому и в целом квантовый алгоритм разложения на множители требует полиномиального числа классических и квантовых операций.

Упражнения

10.1. Состояние $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ (одно из знаменитых «состояний Белла») обладает странными свойствами (кубиты в нём «сцеплены», или «спутаны»). Вот некоторые из этих свойств:

(a) Предположим, что это состояние можно разложить в (тензорное) произведение двух кубитов (см. врезку на с. 300) $\alpha_0|0\rangle + \alpha_1|1\rangle$ и $\beta_0|0\rangle + \beta_1|1\rangle$. На-

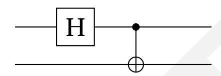
пишите возникающие условия на коэффициенты α_0, α_1, β_0, β_1 (четыре уравнения) и покажите, что они несовместны (и тем самым разложение в тензорное произведение невозможно).

(b) Что произойдёт при измерении первого кубита в этом состоянии?

(c) Что произойдёт, если после этого измерить и второй кубит?

(d) Почему ответ на предыдущий вопрос выглядит удивительно, если два кубита находятся на большом расстоянии друг от друга?

10.2. Покажите, что следующая квантовая схема позволяет приготовить два кубита в состоянии Белла $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, начав с чистого состояния $|00\rangle$: применим элемент Адамара к первому кубиту, а затем CNOT с первым кубитом в качестве управляющего и со вторым — управляемым.



Что будет, если применить эту же схему к состояниям 10, 01 и 11? (Получатся состояния, которые тоже называют состояниями Белла.)

10.3. Что получится в результате применения преобразования Фурье к суперпозиции с равными коэффициентами, то есть к вектору $\frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} |j\rangle$?

10.4. Найдите преобразование Фурье (по модулю M) вектора $|j\rangle$.

10.5. *Свёртка и умножение.* Сдвинем коэффициенты в суперпозиции $|\alpha\rangle = \sum_j \alpha_j |j\rangle$ на l по кругу, получится $|\alpha'\rangle = \sum_j \alpha_j |j+l\rangle$. Пусть преобразованием Фурье состояния $|\alpha\rangle$ является $|\beta\rangle$. Покажите, что преобразованием состояния $|\alpha'\rangle$ будет $|\beta'\rangle$, где $\beta'_j = \beta_j \omega^{lj}$. Выведите отсюда, что преобразованием состояния $|\alpha'\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}} |jk+l\rangle$ будет состояние $|\beta'\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \omega^{ljM/k} |jM/k\rangle$.

10.6. Покажите, что если элемент Адамара применить к обоим входам элемента CNOT, то получится снова элемент CNOT, только управляющий и изменяемый кубиты будут переставлены.



10.7. Элемент «управляемого обмена» (controlled swap, C-SWAP) применяется к трём кубитам и переставляет второй и третий, если значение первого равно 1 (он переводит базисные состояния в базисные, а на суперпозиции продолжается по линейности).

(a) Покажите, что каждый из элементов NOT, CNOT и C-SWAP является обратным к самому себе.

(b) Покажите, как можно реализовать конъюнкцию (AND) с помощью C-SWAP. Какие значения надо подать на три входа элемента C-SWAP, чтобы на одном из выходов получить конъюнкцию двух входов $a \wedge b$?

(c) Как осуществить копирование выхода с помощью этих трёх элементов? (Это означает, что при входах a, 0 надо получить выходы a, a.)

(d) Покажите теперь, что для любой классической схемы C существует квантовая схема Q, использующая только элементы NOT и C-SWAP и эквивалентная C в следующем смысле: если C на входе x даёт выход y, то схема Q даёт выход $|x, y, z\rangle$ на входе $|x, 0, 0\rangle$. (Значения битов z, возникающие в процессе вычисления, могут быть произвольными.)

(e) Постройте квантовую схему Q^{-1}, которая на входе $|x, y, z\rangle$ даёт $|x, 0, 0\rangle$.

(f) Наконец, покажите, как построить квантовую схему из элементов NOT, CNOT и C-SWAP, которая на входе $|x, 0, 0\rangle$ давала бы выход $|x, y, 0\rangle$.

10.8. В этой задаче мы доказываем, что если N является произведением двух простых чисел p и q, а число x — случайно выбранный остаток по модулю N, для которого НОД$(x, N) = 1$, то с вероятностью как минимум $3/8$ порядок r числа x по модулю N чётен и, кроме того, $x^{r/2}$ является нетривиальным квадратным корнем из N.

(a) Пусть p — нечётное простое число, а x — случайно выбранный (равномерно) остаток по модулю p. Покажите, что порядок p будет чётным с вероятностью не менее $1/2$. ((Подсказка: используйте малую теорему Ферма, см. раздел 1.3.))

(b) Используя китайскую теорему об остатках (упражнение 1.37), покажите, что с вероятностью как минимум $3/4$ порядок случайного остатка x по модулю N будет чётным.

(c) Покажите, что если порядок этого остатка будет чётным, то (условная) вероятность события $x^{r/2} \equiv \pm 1$ не превосходит $1/2$.

Исторические замечания и книги для дальнейшего чтения

Главы 1 и 2

Классический учебник по теории чисел:

G. H. Hardy, E. M. Wright. Introduction to the Theory of Numbers. Oxford University Press, 1980.

Алгоритм проверки простоты числа был открыт Робертом Соловеем и Фолькером Штрассеном в середине 1970-х годов, криптосистема RSA была придумана несколькими годами позднее. Подробности см. в книге

D. R. Stinson. Cryptography: Theory and Practice. Chapman and Hall, 2005.

Учебник по вероятностным алгоритмам:

R. Motwani, P. Raghavan. Randomized Algorithms. Cambridge University Press, 1995.

Универсальные семейства хеш-функций были предложены Ларри Картером (Larry Carter) и Марком Вегманом (Mark Wegman) в 1979 году. Алгоритм быстрого умножения матриц найден Фолькером Штрассеном (Volker Strassen) в 1969 году. Им же в соавторстве с Арнольдом Шёнхаге (Arnold Schönhage) был предложен быстрый алгоритм умножения чисел: с помощью некоторой модификации быстрого преобразования Фурье этот алгоритм перемножает n-битовые числа за $O(n \log n \log \log n)$ битовых операций. Чуть более быстрый алгоритм был недавно получен Мартином Фюрером (Martin Fürer).

Глава 3

Алгоритм поиска в глубину (вместе с многими приложениями) был предложен Джоном Хопкрофтом (John Hopcroft) и Робертом Тарьяном (Bob Tarjan) в 1973 году. Их вклад был отмечен премией Тьюринга — одной из высших наград в области информатики. Двухчастный алгоритм выделения компонент сильной связности предложил Рао Косарайю (Rao Kosaraju).

Главы 4 и 5

Алгоритм Дейкстры был открыт в 1959 году Эдсгером Дейкстрой (Edsger D>kstra, 1930—2002). Идею алгоритма нахождения минимального покрывающего дерева можно обнаружить в статье 1926 года, написанной чешским

математиком Отакаром Борувкой (Otakar Borůvka). Анализ структуры данных для непересекающихся множеств (с чуть более сильной оценкой, чем рассмотренная нами оценка $\log^* n$) провёл Роберт Тарьян. Коды Хаффмана названы в честь Дэвида Хаффмана (David Huffman), который изобрёл их в 1952 году (будучи аспирантом).

Глава 7

Симплекс-метод был открыт в 1947 году Джорджем Данцигом (George Danzig, 1914—2005). Принцип минимакса для игр с нулевой суммой открыл в 1928 году Джон фон Нейман (John von Neumann), знаменитый математик и один из разработчиков первых компьютеров. О линейном программировании хорошо написано в книге

V. Chvátal. Linear Programming. W. H. Freeman, 1983.

Учебник по теории игр:

M. J. Osborne, A. Rubinstein. A course in game theory. MIT Press, 1994.

Главы 8 и 9

Понятие NP-полноты появилось в работе Стивена Кука (Steve Cook): в 1971 году он доказал, что задача выполнимости NP-полна. Годом позже Ричард Карп (Dick Karp) доказал NP-полноту двадцати трёх задач (в том числе всех перечисленных в главе 8), и важность этой концепции стала бесспорной (за эти работы Кук и Карп был награждены премией Тьюринга). Независимо от них к тому же понятию NP-полноты пришёл Леонид Анатольевич Левин, тогда ещё находившийся в СССР.

Классическая книга о NP-полных задачах:

M. R. Garey, D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman, 1979. (Перевод: *М. Гэри, Д. Джонсон*. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.)

Учебник по теории сложности в целом:

C. H. Papadimitriou. Computational Complexity. Addison-Wesley, 1995.

Глава 10

Квантовый алгоритм разложения на множители был открыт в 1994 году Питером Шором (Peter Shor). Введение в квантовую механику, ориентированное на приложения к информатике, можно найти на сайте

`http://www.cs.berkeley.edu/~vazirani/quantumphysics.html`

Подробно о квантовых вычислениях см. в записях лекций по курсу «Кубиты, квантовая механика и компьютеры»:

`http://www.cs.berkeley.edu/~vazirani/cs191.html`

Указатель имён и терминов

Санджой Дасгупта
Христос Пападимитриу
Умеш Вазирани

Алгоритмы

Издательство Московского центра
непрерывного математического образования
119002, Москва, Большой Власьевский пер., 11. Тел. (499) 241-74-83.

Подписано в печать 08.04.2014. Гарнитура Charter ITC. Формат 70×10 $^{1}/_{16}$.
Бумага офсетная. Печать офсетная. Печ. л. 20. Тираж 1000. Заказ .

Отпечатано в типографии ООО «ТДДС-СТОЛИЦА-8».
Тел. 8 (495) 363-48-86.
http://capitalpress.ru